DEPARTMENT

OF

COMPUTER SCIENCE

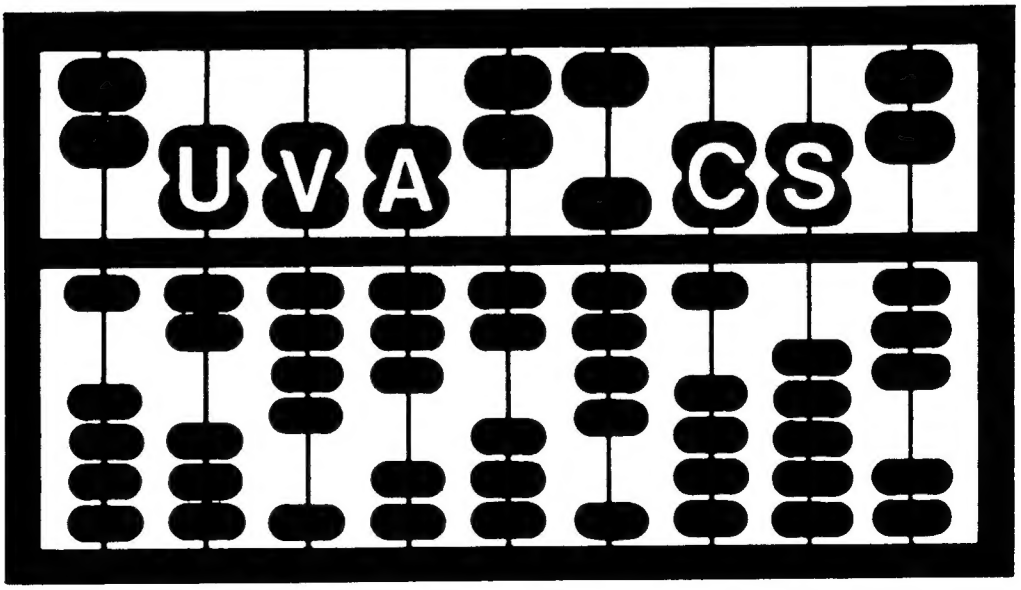
UNIVERSITY OF VIRGINIA

Software for Advanced Vision Systems

Thomas J. Olson
Frank Z. Brill
Glenn S. Wasson
Jennifer A. Wong
Ari S. Rapkin

Computer Science Report CS-94-31
April 13, 1995

CHARLOTTESVILLE, VIRGINIA 22903



School of Engineering and Applied Science

Software for Advanced Vision Systems

Thomas J. Olson
Frank Z. Brill
Glenn S. Wasson
Jennifer A. Wong
Ari S. Rapkin

Computer Science Report CS-94-31
April 13, 1995

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	4/13/95	final report 20 Jun 94 - 2 Feb 95

4. TITLE AND SUBTITLE
Software for Advanced Vision Systems

5. FUNDING NUMBERS
G N00014-94-1-0841
PR c34b020---01

6. AUTHOR(S)
Thomas J. Olson, Frank Z. Brill, Glenn S. Wasson,
Jennifer A. Wong and Ari S. Rapkin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903

8. PERFORMING ORGANIZATION REPORT NUMBER
CS-94-31

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Ballston Tower One
800 North Quincy Street
Arlington, VA 22217-5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The goal of the Advanced Vision Systems program is to develop accelerators that provide a dramatic improvement in performance for applications in image understanding, automatic target recognition, real-time computer vision, and other domains with similar processing needs. This study addresses the question of what types of software support will be required to make the systems useful. After examining the needs of users in the various domains and reviewing available technology, we conclude that the following components are necessary:

1) a library of image processing primitives similar to that proposed for the Image Understanding Environment.
2) compilers and other tools for extending the library.
3) a GUI application builder in the style of KHOROS that allows library routines to be combined into larger applications interactively.
4) a run-time environment that allows complex sequences of operations to execute on the accelerator in parallel with host computation.

The report concludes with a discussion of options for meeting these requirements and a discussion of problems that may arise.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software tools, image processing, automatic target recognition	76
	16. PRICE CODE: unclassified

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Software for Advanced Vision Systems

Thomas J. Olson, Frank Z. Brill, Glenn S. Wasson, Jennifer A. Wong and Ari S. Rapkin

1 Introduction

The goal of the ARPA Advanced Vision Systems program (AVIS) is to develop hardware accelerators that will provide a dramatic improvement in performance for machine vision, image processing, automatic target recognition, and other problem domains with similar processing needs. The accelerators will be used in environments ranging from high-end PCs and workstations to multiprocessors and application-specific embedded systems, and will be capable of taking input from their host environments or from a variety of sensors. AVIS accelerators should result in substantial increases in research and development productivity and in the capability of fielded systems.

In order for the architectures developed under AVIS to be useful in their intended domains, it is essential that they be supported with appropriate software. Among the types of software that will be needed are:

- *runtime environments* that provide flexible communications and control channels between host and accelerator.
- *compilers, debuggers, and performance monitoring tools* that allow programmers to create applications, library routines and higher-level programming tools.
- *software development environments* that allow programmers to combine sequences of primitive operations and use them to build substantial applications.

The goal of this study is to examine the needs of AVIS applications in each of these areas, to identify possible approaches to meeting those needs, and to suggest ways of reducing those approaches to practice. It is divided into four main sections. The remainder of this section briefly describes the motivation and methodology of the study. Section 2 examines the needs of users, programmers and system designers in the various application domains that the AVIS program is intended to support. Section 3 reviews available software technology at all three levels and assesses how well available tools meet the needs of AVIS applications. Finally, section 4 makes recommendations about what types of software support AVIS systems require and how it should be provided on the proposed architectures.

1.1 Motivation

Although the end product of the AVIS program will consist of hardware and software systems, system-building is not the goal of the program. Rather, the goal is to have a substantial impact on research and development in computer vision and related areas, and to enable advanced military and commercial applications. Thus the success of the program will ultimately be measured not by the hardware it produces, but by how widely and effectively that hardware is used.

Software has a critical role to play in making AVIS systems usable. In addition to the bare accelerator hardware, a developer who wants to build an accelerated application will need:

- **a way for the host computer to launch tasks on the accelerator.** This will require providing software that allows the host computer to communicate with the accelerator and control its operation. If the accelerator is capable of general-purpose computation, it may also involve writing an operating system or other executive to handle task dispatching, communications and other housekeeping chores.
- **a way to produce executable code for the accelerator.** The software needed for this task will depend, of course, on the architecture of the accelerators. At a minimum, however, it will include compilers, loaders, and profiling and debugging tools. These tools will make it possible to describe the primitive operations that programmers need to perform (e.g. library routines), and will support writing higher level tools and applications.
- **a convenient way to build large applications.** Although ideally the low-level programming tools should be completely independent of the accelerator architecture, this is unlikely to be possible in practice. It is thus a fact that for many architectures, working at the low level will be significantly more painful than programming a conventional computer in a high-level language. Thus developers will need some sort of higher-level programming environment that minimizes their need to write low-level code. This environment should be architecture-independent, so that applications written with it can be ported to later-generation machines when they become available.

The challenge to AVIS system designers is to provide software tools for each of these roles in a way that satisfies the needs of users across all of the domains of interest. The goal of this report is to provide them with the information they need to do that. The report considers the kinds of computations that users in each domain do, the environment in which they work and the tools they currently prefer. It also describes the state of the art in software technologies that seem relevant to the program. Finally, it makes recommendations about what sort of tools are most likely to satisfy user needs, yet can be built without requiring dramatic advances in software technology.

Much of the information presented here is drawn from current literature in computer vision, software engineering and parallel computation, and is referenced below where appropriate. We have also spent a great deal of time talking with researchers in the domains of interest, trying to find out what they need and want from AVIS software tools. We are very grateful to those who took time to share their experiences and opinions with us. Space does not permit a complete list, but we would particularly like to thank Tom Bannon (Texas Instruments), Ross Beveridge (U. Colorado), Chris Brown (U. Rochester), David Coombs (NIST), Larry Davis (UMD), Ray Rimey (Martin Marietta), Kevin Willey (WP AFB), and Bruce York (Sverdrup Technologies). This work was supported by ARPA/ONR under grant no. N00014-94-1-0841.

2 User Needs

The fundamental goal of the AVIS program is to accelerate the rate of technological progress in machine vision, automatic target recognition, image understanding, and related areas. In order to do this, the program seeks to produce image processing and computing accelerators that allow people working in these areas to do their jobs more quickly and effectively. This section of the study is intended to determine what kind of computations the intended users need to do, and to examine how these needs constrain the kind of software that must be provided with the accelerators in order to make them useful.

AVIS accelerators are likely to be useful to researchers in a wide variety of disciplines, not all of which can be foreseen at the present time. For the sake of conciseness, this study focusses on three specific domains that seem particularly likely to benefit from the AVIS program: image understanding, real-time machine vision, and automatic target recognition. Other domains will be mentioned in passing, but their needs will not be allowed to drive the requirements for the program.

The domains we are focussing on are all active research areas and contain many unsolved problems. Thus it is difficult to make categorical statements about the sort of computations that the domains *per se* require. The recommendations made here are based on the general community consensus in each area about the kinds of computational primitives that are needed and the kinds of paradigms that are effective.

The remainder of this section is divided into three subsections. Section 2.1 discusses the general character of image computation and raises issues that are common to all of the domains of interest. Section 2.2 takes up the domains individually and explains how they relate to the general model. Section 2.3 presents examples of 'typical' computations. We stress that these examples are not broad enough to serve as benchmarks. They are intended to be representative of domain computations, but do not test all of the capabilities that users in these areas require.

2.1 The Character of Image Computation

Although the general problem of analyzing images with a computer remains unsolved, there is a general consensus as to the computational steps and levels of representation that are intrinsic to the problem [5,32]. This consensus has remained stable for more than a decade and is not seriously questioned at present. It provides a useful way of organizing the discussion of the kinds of computations that AVIS accelerators will need to support.

The consensus view is that image understanding involves three fundamentally different types of operations, which we shall here call low, intermediate and high level computations. Each level is concerned with particular types of information and has its own characteristic data structures and computational primitives.

The low level is concerned with computations on images, broadly understood to include two dimensional arrays whose entries or pixels are the projections of properties of the three-dimensional world. In general, any computation that maps an input image to an output image is considered low level. Low level tasks include image operations such as noise removal, filtering, restoration, local feature extraction and statistics-gathering. They also include the computations required to construct images of intrinsic world properties such as depth, reflectance, or motion

from ordinary camera or video images. They rely heavily on understanding of how images are formed: how light interacts with surfaces, how sensors work, where noise is introduced and so on.

At the intermediate level the task is to group pixels in some sort of image into extended structures such as lines, curves, or regions. This segmentation process can be generalized to include such things as grouping pixels in a depth image into 3D surfaces. It depends heavily on assumptions about how the world behaves, e.g. that it is made up of piecewise smooth opaque surfaces.

The highest level is concerned with establishing relations between intermediate level groups and relating them to general knowledge about the world. For example, a high-level task might be to take an aerial photograph and identify those subsets of the lines on it that correspond to roads, vehicles or buildings. Processing at this level requires a tremendous amount of knowledge of the domain; for example, that vehicles are often found on roads or near buildings, but very rarely on buildings. In the limit the concerns of high-level image analysis merge with those of general artificial intelligence.

2.1.1 Low Level Image Computing

Early filtering and image processing operations make up a large fraction of the workload of many image analysis applications. The primitive computations used are well understood and are useful in many domains. In addition, many of them are highly regular and exhibit obvious data parallelism. Thus they are obvious candidates for software standardization and hardware support.

The defining characteristic of computations at this level is that they operate on image data and produce as output either modified images or statistical descriptions of images. For the purpose of talking about computation, the image operators can be grouped according to the data dependencies they exhibit. The major classes of operators are:

- **Statistics operators.** These take in images and produce descriptions of properties over the entire image or region of interest. Descriptions may be based on point samples (for first-order statistics such as mean, variance, or modes (i.e. histogram information)) or neighborhoods (for higher-order measures such as grey-scale co-occurrence matrices).
- **Pixelwise monadic operators** produce an image whose pixel values are pure functions of the corresponding input pixel. Examples include histogram transformations, thresholding, squaring and so on.
- **Pixelwise dyadic operators** produce images whose pixel values are functions of corresponding pixels in two images. Pixelwise addition and subtraction are very common. More general examples include such things as converting images of X and Y intensity derivatives to polar form.
- **Monadic neighborhood operators**, called 'stencil algorithms' in the parallel processing literature, produce images whose pixels are functions of a local neighborhood in the input image. The convolution operation, in which each output pixel is a weighted sum of the corresponding input neighborhood, is the largest single consumer of cycles in many image processing applications. Morphological operators and rank (median) filters also follow this pattern, as do miscellaneous non-linear computations such as zero crossing detection. Many relaxation computations (e.g. optimizations based on regularization or Markov random field models) do so as well, but may require auxiliary images for such things as line processes.

- **Resampling and data movement operators.** Image processing programs often spend a surprising amount of time simply moving pixels around in memory. Trivial examples include extracting rectangular windows from an image or combining multiple windows to form a composite image. More sophisticated applications may need to change the resolution of an image by subsampling or interpolation. Important special cases include the computations that form Gaussian [8], wavelet [31] or other [11] pyramids. Some applications require the ability to perform affine, polynomial or arbitrary warping.
- **Orthogonal transforms** such as the Fourier or Walsh-Hadamard transform are critical to SAR image formation and to many signal restoration and registration algorithms. In these transforms, each output pixel is a function of every input pixel. Fortunately the communications patterns required are highly regular, and efficient parallel algorithms exist.

Because there is wide agreement about the sorts of low-level image computations that are useful, there are a number of standards describing specific sets of operators. An example is the PIKS standard, discussed below. Table 2 on page 36 contains a list of the PIKS operators, which can be compared to the classification given above.

2.1.2 Intermediate Level Computations

Intermediate level image computations generally take images and other information as input and return a description of the image in terms of extended features such as lines, curves, or regions. These features may be assembled into perceptual groups that are related under some criterion; for example, a group of lines may form a closed figure. Regions may be organized into a hierarchy, with each region divided into subregions. Intermediate level computations involve frequent reference to image data, and often involve a substantial amount of search. Thus they may require a great deal of computation even for relatively simple applications.

Intermediate level computations are less easy to parallelize than low level computations. Because intermediate level features can extend across the whole image, the algorithms that extract them require non-local communications. The number of features is typically not known a priori, and features are often distributed non-uniformly across the image. This means that data partitioning and scheduling decisions must typically be made at run time if the hardware is to be used efficiently. Finally, there is only weak consensus among users as to what computations are important. Thus it is unclear what algorithms should be standardized and included in applications libraries.

Because intermediate level computations are less standardized than low level computations, it is not possible to give an exhaustive list or taxonomy of the methods in current use. Instead, we present a sampling:

Line and curve finding seeks to produce chains of image locations corresponding to logical boundaries in the image. The boundaries may be defined by a parametric model (e.g. lines, circles or ellipses), may be arbitrary space curves, or may combine characteristics of both (e.g. splines). Extraction typically begins by identify points that are likely to lie on a boundary. This is done using low level operations such as gradient or zero crossing detection. The intermediate level computation consists of grouping boundary points into sets that fit one or more instances of the

model. Grouping methods include the Hough transform, edge following, clustering based on orientation, or iterative relaxation processes (snakes [22]).

Region finding is in some sense the dual of boundary finding; pixels are grouped into sets based on their fit to a model of region properties. The model may be as simple as an assumption of uniform brightness, or may involve elaborate models of local statistics (texture) as well as models of corresponding world surfaces. In the past decade the trend has been toward algorithms that use three-D models of world surfaces as the driving constraint for grouping pixels together. For example, a system might initially group pixels based on proximity and similarity of motion, then estimate rigid motion parameters for each group, and then refine the grouping by discarding locations that are poor fits to the solution for their region. The general computational paradigm remains the same, however; pixels are grouped based on their fit to a dynamically evolving model of the regions present in the scene.

2.1.3 High Level Computations

High level image computations match extended features extracted from images to relational models of objects. For example, they might try to match a collection of observed surface patches and edges to a CAD model of, say, a particular type of airplane. In the general case the problem is to establish correspondences between a subset of the scene features and a subset of the model features. The task is complicated by the geometric degrees of freedom of the problem and by noise and occlusion, and is expensive because the combinatorics of search are inherently unfavorable.

High level computations are hard to parallelize. This is partly due to the fact that good search algorithms involve a great deal of constraint propagation and hence require a lot of communication between tasks. As in the case of intermediate level vision, the computational load shifts dynamically and makes partitioning and load balancing difficult. Finally, there is little consensus about what algorithms are best for any given application.

Paradigmatic high level tasks are hard to come by, but the interpretation tree match algorithm of [16] is typical of the type of computations that most algorithms require. In this algorithm a set of observed features is matched to a model one feature at a time, using geometric constraints to prune the search space. In a 2D matching problem, for example, after deciding to associate a model line segment with a particular scene line segment, the algorithm can infer the scaling and rotation components of the transformation. This allows it to reject any subsequent feature associations that would imply different scaling and rotation parameters. This paradigm was originally designed for matching with precise CAD models, but can be generalized almost indefinitely. For example, the well-known functional recognition algorithm of [50] is also a tree search with constraint propagation. However, their constraints are based on how an object is used rather than on a geometric model. For example, a chair must have a surface called its seat that is roughly parallel to the floor, that is neither too close nor too far from the floor, and that has enough clear space above it to contain a human torso.

2.2 Specific Domain Requirements

The previous section presented a generic discussion of the types of computations that are performed in the domains under study. In this section we turn to the question of how the individual domains differ in their use of these computations. Our primary conclusion is that the domains dif-

fer less than expected in the types of computations they require: all of them perform low, intermediate and high-level operations and use similar algorithms at each level. There are some differences in the job mix and in the degree of precision required in the results, but these distinctions are relatively minor. The major differences between domains are in the hardware and software environments in which the computations are performed, and in the source and form of the image data they manipulate. In this section we will focus on these differences and discuss their influence on AVIS software requirements.

2.2.1 Image Understanding

The first domain to be considered is image understanding. This area has been the subject of an ongoing ARPA research program (see e.g. [1]) covering a very broad range of applications. Here we will use the term in a more restricted sense, excluding such topics as automatic target recognition (which will be discussed separately below). By 'image understanding' we mean the problem of analyzing image data for the purpose of extracting as much information as possible about the scene, in a manner which is as independent as possible of the use to which the information will be put. For example, research directed to recovering a 3D model of an object from multiple uncalibrated camera views of the object would fall into this category. The common thread in these applications is that they are all concerned with building detailed models of the objects in the scene. Accuracy of the description is the primary criterion for success, and the primary activity of people working in this field is designing newer and more accurate algorithms for building these descriptions. These algorithms may operate at any of the three levels described above, and emphasize the most elaborate and complex algorithms in each class.

Since image understanding work is highly experimental, the tools and environments used by people working in this area are designed for the convenience of the researcher. This translates in practice to support for a) constructing and debugging new algorithms and b) applying the algorithms and analyzing their results. The computational environment is typically a large workstation with a high-quality display, perhaps with network access to large-scale storage facilities and/or special computational resources such as supercomputers.

The dominant computational paradigm for researchers in image understanding is the conventional single-threaded program in C, LISP or other standard language. Since speed is a secondary concern, researchers optimize for ease of development, which dictates the use of familiar languages and computational paradigms.

Individual image understanding projects tend to be concerned with some small part of the general image understanding problem, e.g. designing a better early vision operator, feature extractor or matching algorithm. Thus researchers typically maintain collections of separate programs for the various standard early and intermediate-level processing steps. When they are working on a new high-level algorithm, for example, they are likely to apply the required early and intermediate-level preprocessing to a set of test images and save the results. They can then apply various versions of the algorithm under development to these saved descriptions, avoiding the cost of repeating the preprocessing.

2.2.2 Real-time computer vision

The domain of real-time computer vision includes any application in which image processing and computer vision techniques are embedded in a control system. Only recently have computers and systems become powerful enough to enable significant progress in this area. Among the driving applications are robot manipulation and navigation, autonomous vehicle guidance, and modelling of human oculomotor behavior.

Real-time vision problems are of interest for more reasons than the potential usefulness of the applications. Embedding computer vision techniques in a control system turns out to change the character of the computation in a fundamental way [52]. The real-time constraint means that systems must balance the usefulness of more information against the cost of extracting it, and must make intelligent decisions about what aspects of the scene are worthy of attention. Real-time systems with autonomy also have the option of perturbing the imaging situation (e.g. by moving) to improve the quality of their input data. In general, real-time systems can take advantage of their perceptual history and goals to reduce the amount of computation they must perform.

Computational constraints

Real-time computer vision systems must contend with a number of problems not faced by more general image understanding systems. The most obvious problem is the input data rate. A single black-and-white video camera produces roughly ten million eight-bit pixels per second. A stereo system produces two such data streams, and a high-resolution color camera three. High-resolution cameras, FLIR sensors and so on can produce substantially more. Depending on configuration, then, an autonomous robot may need to process anywhere from ten to a hundred million pixels per second.

The problem of data rate is compounded by the need to process it with very low latency. Delay severely limits the bandwidth of control systems and requires special techniques (e.g. predictive methods) to maintain stability. Researchers working on optical tracking have found that processing delays must be kept to the order of a single video frame time in order to achieve anything remotely like human-level performance.

Another problematic aspect of real-time computer vision is the need for predictable running times. If delay is a problem for control systems, unpredictable delay is a disaster. Experimenters go to great lengths to make the running time for data acquisition and early processing as free as possible from data dependencies and other sources of non-determinism. It is worth noting, though, that researchers recognize that high-level computation inherently involves unpredictable running times. Finding ways to tolerate this is a focus of current research.

The result of these constraints is a style of computation that is quite different from that found in more conventional computer vision programs. Vision is no longer seen as a function that maps input images to some sort of symbolic description. Instead, researchers think of vision systems as collections of processes, each with their own parameters, internal state, and processing rate. Some processes may be essential parts of the system that are always present: examples might include 'looming' detectors or eye movement controllers. Others may be created to answer specific task-related questions about the scene, and will be removed when their job is done or the system's priorities change. For example, a reconnaissance robot might create a task to scan an airstrip and label things that look enough like helicopters to be worthy of a closer investigation.

Real-time vision systems tend to include a diverse collection of hardware and software that makes for a very difficult development environment. The overall system might include a UNIX workstation for planning and reasoning, an attached tolerator for digitization and image processing, and a number of board-level real-time computers to perform inverse kinematics and motor control computations. Frequently each component of the system has its own operating system and development environment. Programmers spend a great deal of time trying to get the various parts of the system to communicate reliably and quickly enough to make the control loops work properly.

In the past, the need to control latency and the emphasis on low-level tasks such as gaze control have led real-time vision researchers to emphasize simple, low-level image processing operations such as filtering and connected component labelling. There is increasing interest, however, on finding ways to integrate recognition and other high-level tasks into real-time vision systems. The trend is toward systems containing multiple processes (typically running on multiple CPUS), some of which run tight, fast control loops while others execute less predicable high-level tasks.

2.2.3 Automatic Target Recognition

Automatic target recognition systems use the technologies of image understanding, signal processing and related fields to identify image locations corresponding to specific objects or targets. Input to an ATR systems may include SAR or FLIR images as well as conventional intensity images, and may be augmented with symbolic encodings of other sensor reports, intelligence, doctrine of the opposing forces et cetera. Typical ATR applications include finding and classifying armored vehicles in high-altitude SAR images, or recognizing high-value targets in a naval battle group.

The methods used by ATR systems fall within the domain of image understanding, but the very specific nature of the ATR mission results in a slightly different computational paradigm. For ATR systems, only targets or potential targets are interesting; everything else is clutter. ATR systems do not need to understand the scene as a whole unless doing so makes it easier to locate targets. Thus their early processing is often intended to filter out image locations that are not worth looking at.

Many ATR computations can be divided into phases of preprocessing, detection, and classification [6]. Preprocessing operations are intended to remove noise and imaging artifacts that may be confused with target features. SAR and FLIR sensors in particular are subject to characteristic artifacts that may create spurious texture or edges in images. Operations used include filtering, gain correction, and contrast enhancement.

The detection phase of an ATR computation attempts to identify image locations that may correspond to targets. The general strategy used is to apply global feature extraction operations (e.g. edge detection, template matching, or non-linear operations such as texture measurement) to find features that are correlated with target presence. Since detection operations are applied everywhere they are generally fast and simple. They are intended to reduce the search region to which the more expensive classification operations must be applied.

The classification phase compares detection outputs against models of known targets and either labels them with a target type or discards them as not of interest. In early ATR systems this

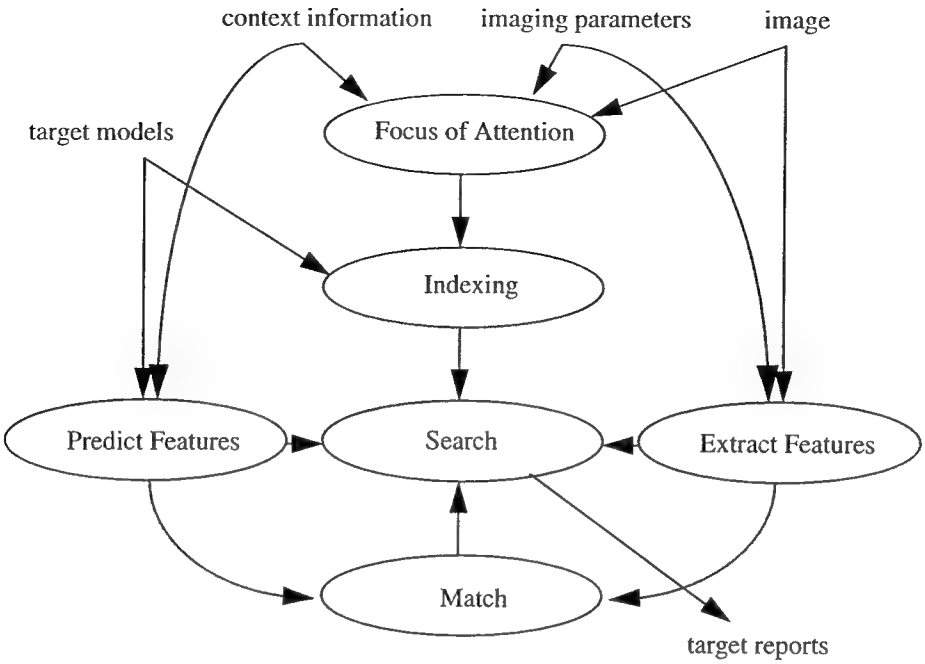


FIGURE 1. An architecture for model-driven ATR, after [45].

was often done by comparing signatures (feature vectors) using the techniques of statistical pattern recognition. These methods break down when the set of possible appearances of a target becomes large, because the number of target models that must be stored and compared becomes intractable. Unfortunately in realistic situations the number of appearances is nearly always large. For example, the appearance of a vehicle depends on viewing direction, occlusion or camouflage, configuration (hatches and doors open or closed etc.), presence or absence of shadows, or operational state (e.g. for FLIR, engine on or off). Thus modern ATR systems are increasingly being designed around detailed models of target variability and image formation.

A system-level description of a generic model-based ATR system is provided by the report of the ARPA/SAIC system architecture study group, convened at the request of ARPA/STO in connection with the War Breaker program [45]. The report describes an architecture for analysis of single high resolution SAR images, but the paradigm is general and should apply to many other ATR situations. The inputs to the system are assumed to consist of the image, image information such as imaging parameters, and contextual information (maps, site models, target models, and any prior information from previous analyses). The output of the system consists of target reports with confidences, plus useful information such as terrain classification.

Figure 1 (after [45]) shows the general structure of the system. The functions labelled Focus of Attention and Indexing correspond loosely to the detection phase discussed above, while the Predict, Extract, Matching and Search functions make up the classification phase. The specific tasks of the functions are as follows:

Focus of Attention: The FOA function takes images, image parameters and collateral information as input. Its output is a list of regions of interest (ROIs). Each ROI is annotated with the system's confidence that targets are present, a description of the target and surrounding clutter,

and any preliminary feature extraction or segmentation results. In addition the ROI may contain information that registers it to any available maps or site models.

Operations performed by the FOA function include

- preprocessing: linear and non-linear filtering, morphological filtering
- registration: line and feature extraction, image-to-image or image-to-model registration
- segmentation and labelling: MRF optimization, pixel classification
- target detection: thresholding, CFAR detection, preliminary classification based on shape, texture etc.

Indexing: The inputs to the Indexing function are the ROIs produced by the FOA function plus a set of target models processed to extract indexing features. It produces a list of hypothesized target labels, aspects, states and configurations that might give rise to the observed features. The intent is to generate all of the possibilities that are worth exploring in more detail. Thus probability of detection is favored heavily over probability of false alarm, and mutually exclusive hypotheses are tolerated.

Components of the indexing task include estimating the target aspect, deciding what specific target types are likely, and associating confidences with the hypotheses. The methods are typically based on comparing image features (e.g. by normalized correlation or some sort of non-linear template match) against a library of signatures that coarsely sample aspect space. Confidences are essentially measures of match quality, but may take into account contextual information or confusability with other targets or clutter.

The general method used for target classification is a cycle of hypothesizing target parameters, predicting the target appearance that would result, and matching the prediction against features extracted from the image. The overall classification task is under control of the Search function. The ARPA/SAIC report identifies rule-based systems embodying heuristics derived from human experts as the dominant technology for this task. The goal of the Search function is to perform the relatively expensive classification task in an efficient manner.

Feature Extraction: The feature extraction function examines the image and attempts to locate image features that match predictions based on the current hypotheses. It is invoked by the search module and supplied with an ROI to work on and a description of the feature to be sought. It returns numerical values and certainties for the features it was asked to find. For SAR, features might include bright peaks in a particular region, or areas of texture or shadow of a particular shape.

Prediction: The prediction function takes a hypothesis consisting of a target model, state information, and information about the imaging situation and predicts what features should be visible given the hypothesis. Features may be as general as shape parameters or as specific as a ray-traced rendering of the hypothesized target.

The prediction computation depends heavily on the form of the target model base and the type of features to be predicted. Constructing and validating the model base is a major task in itself, though presumably it can be done off-line. Techniques used may include simulating the image formation process or selecting stored example data from a database.

Matching: The role of the matching function is to provide a numerical estimate of the consistency between the predicted and extracted image features. Ideally this will be in the form of a

likelihood, i.e. probability of the observation given the hypothesis. Performing the task involves examining possible correspondences between predicted and observed features, making allowance for the possibility of occlusion. The final step is to generate a probability. The favored technique at present is evaluation of a Bayesian network.

Development and execution environments

ATR systems vary greatly in their scale and their need for real-time response. ATR systems for missile target selection may need to be physically small, operate on a tight power budget, and respond within milliseconds. At the other extreme, systems for analyzing high-resolution SAR images may be distributed across a heterogeneous collection of MIMD multicomputers, dedicated image processing engines and workstations, and may run for minutes or hours. In general ATR researchers are more willing to invest effort in coding for unusual hardware configurations than are most mainstream image understanding researchers. They tend to build complete end-to-end systems and test them in fairly realistic situations, and therefor must run their systems on hardware that can meet the throughput requirements of the application.

The ARPA/SAIC system architecture described above is intended in part to serve as a blueprint for dividing model-based ATR systems into independent processes. The intent is to provide formal definitions for the interfaces between the modules shown in figure 1, and to let different contractors implement different parts of the system. The architecture would thus simplify systems integration, and perhaps allow system builders to experiment with multiple versions of (say) the indexing module to see how they interact with downstream processing. It would also provide an immediate avenue for obtaining coarse-grained parallelism in ATR applications.

Many ATR groups have adopted the Khoros image processing toolkit [24] as their standard development environment, and many ATR research contracts specify Khoros modules and workspaces as deliverables. There is thus a great deal of interest among ATR researchers in making AVIS development tools compatible with Khoros. Khoros will be discussed below in section 3, and we will return to the question of Khoros support in section 4.

2.3 Sample Tasks

In this section we present a small group of sample tasks that are representative of computations in the AVIS task domains. Our purpose is to give a more concrete sense of the types of computations that AVIS programmers need to perform, and to provide a basis for evaluating some of the development tools to be presented in section 3.

Selecting a set of representative tasks turned out to be remarkably difficult. The first problem is that there really is no such thing as a "typical" vision application. Although most of the computations described in the literature fit one standard patterns or another, the number of standard patterns is very large. It is very rare to find a single program that is representative of more than a small fraction of the domain. The second problem is that descriptions of algorithms in the literature are typically sketchy, and are narrowly focussed on the novel aspects of the algorithm under discussion. It is rare to discuss an end-to-end system in any detail. This is fine for purposes of academic publication, but makes it relatively hard to characterize the computational requirements of the application without a great deal of domain knowledge. Finally, for many ATR applications there are proprietary and, perhaps, security concerns. ATR contractors are often very reluctant to provide details of the computations their systems perform.

We wish to be very clear about the intended role of the tasks described here. They are in no sense a benchmark against which software tools can be measured. Although we have tried to make them broad in scope, they do not exhaust the space of application requirements. They are simply points in the design space, intended to serve as examples of the style of computing that AVIS software tools should be able to support.

2.3.1 The Image Understanding Benchmark (IUB)

The DARPA image understanding community has created a series of benchmarks for evaluating image understanding architectures. Our first example task is the 1988 version of the benchmark [60]. The task in this benchmark is to recognize an object consisting of colored rectangles obeying a particular set of geometric constraints. The input consists of registered (synthetic) intensity and depth images of an object in a field of distractor rectangles, some of which may occlude parts of the object. The program is also given a database containing ten reference objects.

We were initially reluctant to use the IUB task as an example, simply because it is already well known to the IU community. However, the IUB has two advantages that are unmatched by any of the other tasks we considered. First of these is its breadth: the IUB uses a broad set of algorithms at low, intermediate and high levels of the image analysis hierarchy. The algorithms were defined by experts in the field for the express purpose of spanning the space of image computations and stressing parallel architectures to their limits. The second major advantage of the IUB is that it is very well specified. The IUB defines the computations to be performed in great detail, and provides a complete uniprocessor implementation of the task which can be run on any architecture that supports the C language and the standard C library. The IUB definition and uniprocessor implementation are available for anonymous ftp at cax@cs.umass.edu, directory / pub/IUstatic_benchmark.

The IUB task does have weaknesses as well as strengths. It is unfortunate that the task is defined entirely on synthetic images. Using synthetic images allowed the benchmark designers to supply programs that generate arbitrary numbers of random test cases. To some extent, however, it distorts the IU problem by presenting a task which is inherently easier than that of recognizing real objects in real scenes. The IUB also avoids the problems of registering the depth and intensity images, correcting for lens distortion (there is none), and dealing with the calibration issues that plague real-world vision systems. The task does not require any shape or depth estimation from the intensity image, and simplifies the model-matching problem by limiting the model aspect parameters to four degrees of freedom (XYZ translation plus rotation about the viewing direction) instead of six. A final criticism of the IUB is that it is old, and does not include techniques that have become popular since it was designed.

The IUB task in detail

The IUB task is described in great detail in the paper [60] and in the reference materials supplied with the sequential implementation. Here we will summarize the computation in just enough detail to serve as a basis for tool evaluations in the next section.

The basic task performed by the IUB programs is recognition of a 2-1/2D object or 'mobile' consisting of rectangles of various sizes, brightnesses, two-dimensional orientations and depths. The rectangles have fixed spatial relationships given by links in the model. A database of ten mobiles is provided and any one can be in the image. The goal is to determine which of the models is

present, to label image features with the model features to which they correspond, and to update the model with positional data that has been extracted from the images.

Target models consist of a set of rectangles in three-space, each with nominal depth, size, intensity and x/y orientation. The rectangles are connected by invisible links that form a tree structure. To generate a test scene, one of the models is translated, rotated and then embedded in a field of distractor rectangles. Any model rectangle may be partially or fully occluded by a distractor. The model parameters (including link parameters) are then perturbed with Gaussian noise truncated at $\sigma = 3$. Since the links form kinematic chains, link perturbations are cumulative. Thus two rectangles at opposite ends of a long chain impose only loose constraints on each other's positions.

Two input images are generated from each test scene. One is a depth image from an idealized range scanner. It consists of a 512x512 array of 32-bit floating point values, corrupted with additive Gaussian noise. The other is a 512x512 array of bytes representing intensities. The intensity image is noiseless and each pixel color is computed by projecting its center to a unique scene point. Thus rectangle boundaries are subject to aliasing but are clearly defined. Both images are produced by orthographic projection and are perfectly registered.

Preprocessing

The benchmark description specifies the following preprocessing operations for the intensity image:

1) Label four-connected sets of homogenous pixels with a unique integer.
2) Traverse the boundaries of the sets and compute their k-curvature for some specified k. Smooth the k-curvature estimates along the boundary by convolution with a 1x7 Gaussian mask, and find zero crossings of the first derivative of curvature.
3) Label corner points along each boundary. Corners are points that correspond to zero crossings of the first derivative and have curvature exceeding a specified threshold.

For the depth image the specified preprocessing is:

1) Smooth with a 3x3 median filter.
2) Estimate the gradient magnitude using the Euclidean magnitude of responses to the 3x3 Sobel masks.
3) Apply a supplied threshold to the gradient magnitude to obtain a binary image that encodes the positions of depth edge pixels.

Hypothesis generation

The next step in the benchmark task is to identify good candidates for rectangles. The required set of operations is:

1) For each connected component in the intensity image, compute the convex hull of the corner points.
2) Apply geometric tests to see if the component can be interpreted as a rectangle for which at least three of the rectangle corners are visible.
3) For components which pass the test, create a rectangle hypothesis that records rectangle center, lengths of major and minor axes, orientation and intensity.

Once likely rectangles have been extracted, the benchmark task looks for approximate matches between groups of image rectangles and the models. The rectangles recovered from the images can be thought of as forming a complete graph with a link between each pair of rectangles. In order to find good interpretation hypotheses, the task seeks isomorphisms between subgraphs of each model and subgraphs of the scene.

The method used for finding subgraph isomorphisms is essentially similar to interpretation tree matching. Individual scene rectangles are first matched to model rectangles based on intensity and size. This process yields all single-node isomorphisms. Each isomorphism is then extended by making predictions based on the links for the model rectangle, and forming associations between pairs of scene rectangles that are mutually consistent given the model. Each association gives rise to a hypothesized pose for that model in the scene. Pairwise links to the same rectangle that give rise to the same pose are merged to form a single higher-order hypothesis.

The hypotheses resulting from the above step are inherently consistent with the intensity image, but may conflict with the depth image. The next step is to probe the depth image and see if the relative depths as well as the intensities and relative positions of rectangle sets is consistent. Inconsistent hypotheses are discarded, while consistent hypotheses are assigned a match strength.

If a hypothesis requires the presence of a rectangle that was not found in the initial candidate rectangle extraction, the system performs a top-down search for it in the thresholded gradient magnitude of depth image. The search consists of a Hough transform applied to the window of the image that is predicted to contain the rectangle. If the Hough transform detects three of the predicted four depth edges, the rectangle is accepted and added to the hypothesis. If not, a match strength is computed or the hypothesis is rejected as appropriate.

The final result of the above process is a list of hypothesized instances of models in the images. Each hypothesis is assigned an overall match strength, which is the average of the match strengths for its constituent rectangles. The model with the best match strength is returned as the system's interpretation of the scene.

2.3.2 Software Library for Appearance Matching (SLAM)

SLAM is an object recognition system created by Shree K. Nayar and others at the University of Columbia. It is based on the use of principal component analysis to define k basis images that approximately span the space of possible appearances of an object, given a large training set. Each training view of an object can then be represented by the k-tuple which is its projection on the basis set. Interpolating the training set yields a manifold in k-space whose dimensionality is equal to the number of degrees of freedom in the viewing parameters used to generate the training set. Recognizing an object then reduces to finding the manifold that it is closest to. The position of the closest point on the manifold can be used to estimate the viewing parameters, yielding (for example) pose as well as identity.

We selected SLAM as an example task for the following reasons. Like the IUB, it performs a complete end-to-end recognition task, and a complete and portable implementation is available (send email to `slam@cs.columbia.edu` for information). Unlike the IUB, SLAM is based on real images. It relies heavily on techniques that are absent from the IUB, including numerical methods (i.e. solving eigensystems) and fitting B-splines of one, two or three dimensions. It is fundamentally a pattern classification system, and as such resembles many older ATR systems.

SLAM specifics

SLAM consists a collection of programs that can be run from the command line. Many of the tools are also available in GUI form. The underlying routines form a C++ library with a clean API that could be used to implement stand-alone applications, or to replace the supplied implementations with accelerated versions.

SLAM can be logically divided into two tasks. The visual learning task takes sets of preprocessed training images, computes the basis set, and constructs a manifold for each object. The appearance matching task takes a new image and identifies it by determining which manifold it is closest to.

The visual learning task consists of the following steps:

1) Prepare a set of features vectors that can be used for principal component analysis. In the simplest case the pixels of an $m \times n$ image patch can simply be concatenated into a vector of length mn. However, recognition becomes easier if obvious sources of variance are removed from the training set. SLAM provides a GUI image manipulation module that permits interactive preprocessing of the training images. Supported include brightness normalization, segmentation, and size normalization. Images may also be smoothed, subjected to first derivative or Laplacian filtering, or Fourier transformed. The operations required are standard low-level image processing routines: convolution, windowing, statistics and rescaling.

2) Compute principal components of the input image set to obtain an orthogonal basis for representation of the images, and truncate the basis to obtain the eigenspace. SLAM provides both GUI and command-line interfaces for this step. The user can specify files in which to store this information, and also the amount of memory to be used in the computation.

   The method used to compute principal components is proprietary. The basic computation is:

   - compute the mean of the training set and subtract it from every entry, producing a zero-mean sequence.
   - form the matrix M whose columns are elements of the training set, and compute the covariance matrix $C = XX^T/N$, where N is the size of the training set.
   - compute eigenvectors of C and sort by eigenvalue. The eigenvectors are the principal components of the training set. Any standard eigenvector method can be used; see [40] for examples.

   In the case where the training vectors are pixels of an image, the dimensionality N of the training vectors may be larger than the number M of training images. In that case the method described above may be hopelessly inefficient, and the following method (from [55]) is preferable:

   - form a zero-mean sequence and matrix X as above.
   - form the $M \times M$ matrix X^TX/N and compute V, the matrix of its eigenvectors.
   - form the matrix XV. Its columns are the principal components of the training set.

   This method works because if v is an eigenvector of X^TX/N, then $X^TXv/N = \lambda v$. Multiplying by X gives $(XX^T)Xv/N = \lambda Xv$. Clearly Xv is an eigenvector of the covariance matrix XX^T/N and the eigenvalue is unchanged.

3) Project the input images to eigenspace to obtain a set of discrete points. This involves taking the dot product of each training image with each eigenvector found in the previous step.

4) Interpolate between these points to obtain a manifold. SLAM uses a quadratic B-spline for interpolation. It can be resampled at higher frequency to obtain a dense set of discrete points. If the user has computed manifolds for a number of different vector sets, all in separate eigenspaces, it may be desirable to combine them into one eigenspace via Gram-Schmidt orthogonalization (which constructs an orthogonal basis from a set of independent vectors). The Gram-Schmidt rotations can be applied to the manifold points to transform them as well, or the manifolds can be computed after orthogonalization.

   The program checks the input set to determine the dimensionality of the B-spline (curve, a surface, or 3-parameter hypersurface). The B-spline is computed with an open uniform knot vector duplicated at the ends so that the manifold passes through the endpoints of the point set.

   Interpolation, resampling, and orthogonalization are all possible via SLAM modules. The `xmanifold` module allows visualization of the manifolds and projections, and the user can save the output to disk and/or obtain a postscript copy.

The appearance matching code accepts the manifolds and eigenvectors computed above, and performs the following steps:

1) Project the image to be matched into eigenspace, obtaining a single n-dimensional point. Again, this involves a series of dot products.

2) Use a user-selected search strategy to find a point on the manifold closest to the projection of the image into eigenspace. The index of the manifold is the best matching object, and the position on that manifold gives the viewing parameters.

   Rather than use analytical methods to find the actual closest point, SLAM resamples the manifold to produce a dense set of points, and then searches for the closest point in the set. The user can provide a search algorithm, or use those offered by SLAM. These are available through C++ as methods. The provided search methods are:

   - Exhaustive search: compute the distance to every point in the resampled set.
   - Heuristic search: inspect only those manifold points whose first coordinates are "close" to that of the input point. Since the points have maximum spread in the first dimension, this trims the search space considerably. However, success depends on the chosen threshold for "closeness".
   - Binary search: perform binary search separately in each dimension. Improves performance from $O(kn)$ to $O(k\log n)$

2.3.3 Wong's hierarchical tracker

Our third sample application is a real-time tracking system created by one of us (Jennifer Wong) at the University of Virginia. The goal of the system is to find and track human faces in live video input. It is thus an example of a typical real-time vision application. Like many real-time vision systems, the tracker is implemented on special hardware and is carefully tuned for our laboratory environment. Portable C versions of the major routines will be available for anonymous ftp via http://uvacs.cs.virginia.edu/~vision.

The input to the tracker is presumed to be an unbounded sequence of video frames delivered one frame at a time. The output at any given time is the current estimated (x,y) position of the target in the image. In our implementation the system displays the live video stream on a monitor, and overlays a graphical marker on the target. If desired, the target position estimate can also be used to servo the camera to keep the target centered.

The tracker consists of two logically independent processes. The low-level process searches the input image near the estimated target position, looking for the best grey-level match to a patch extracted from the previous image. It uses coarse-to-fine search in a Gaussian pyramid to make the search efficient. The result of the search is a new estimate of target position. The high-level process examines a larger neighborhood around the target position, searching for features that fit its model of a face. If it finds one it reports its position, otherwise it reports failure.

The two tracking processes are intended to be as asynchronous as possible. They communicate only in that both have access to the video stream and can read the estimated target position atomically. In our implementation the nature of our image processing hardware makes it impossible to implement the trackers as separate threads. Instead, the high-level process code is decomposed into six routines of roughly equal running time. The system performs one step of the low-level process and one of the six high-level routines on each pass through the main loop. The result is that the high level process runs at one-sixth the frame rate of the low-level process.

Tracker details

The Wong tracker consists of two logically independent tracking processes that communicate with a control process. Each process has its own model of what constitutes a target, and is capable of generating control signals without help from the other process. The processes cooperate to take advantage of their individual strengths and track more robustly than either task can do on its own.

Each of the tracking processes runs a tight loop in which it accesses the current input image, estimates the target position, and informs the control process of its estimate. The control process combines the results of the two estimates and updates the global estimate of target position. In our implementation it also draws a box around the current target position in the overlay plane of our video display, as an aid in assessing the performance of the system.

The Low Level Tracker: The low-level tracking (LLT) process estimates the tracking error by extracting a patch from the current image centered on the current estimated target position. It compares that patch to neighboring patches in the next image, searching for the one that minimizes the sum of squared pixelwise differences (SSD). It reports the difference in position between the best match and the current target position as its estimate of the error.

At the beginning of the main loop the LLT algorithm has access to a four-level Gaussian pyramid computed from the current image, plus the current estimated target position. Upon arrival of a new image it performs the following steps:

1) Form a four-level Gaussian pyramid from the new image. Level zero of the pyramid is the original image. Level i is obtained from level $i-1$ by convolving with a 5×5 Gaussian mask and subsampling by a factor of two in each direction. The computation can be done efficiently by separating the mask into 5×1 and 1×5 convolutions, and by evaluating the convolution only at points that are needed for the subsampled output. The resulting pyramid consists of four copies of the image at resolutions 512×512, 256×256, 128×128, and 64×64.

2) Extract a four-level *target pyramid* from the current image pyramid. The target pyramid consists of four images centered on the current estimated target position, each extracted from the corresponding level of the current image pyramid. The sizes of the target patches are 64×60, 32×20, 16×14 and 8×6. Note that although the target position at level zero of the pyramid (i.e. the original image) is given by integer image coordinates, the positions of the patches at higher levels pyramid levels may not be. For example, if the current target position estimate is $(23, 47)$, the target patches will be centered at $(23, 48)$, $(11.5, 24)$, $(5.75, 12)$, and $(3.875, 6)$ in their respective pyramid levels. To account for this, bilinear interpolation is used to extract the target pyramid from the current image pyramid.

3) Extract a four-level *search pyramid* from the new image pyramid. The process is analogous to that used to build the target pyramid in step 2, except that the search pyramid is larger by a border of width one pixel all the way around: the levels are 66×62, 34×32, 18×16 and 10×8.

4) Find the best match for the target pyramid in the search pyramid by coarse-to-fine search. The process is follows. Let (e_i, e_j) be the current error estimate, initially $(0, 0)$. Starting at the coarsest level of the pyramid, compute the SSD (sum of squared pixelwise differences) between the target level and nine possible offsets of the search level, i.e.

$$\text{SSD}(i,j) = \sum_{p,q \in patch} (\text{Target}(l,p,q) - \text{Search}(l, p+i+e_i, q+j+e_j))^2$$

for all i, j in $\{-1, 0, 1\}$. Let E_l be i_l, j_l minimizing $\text{SSD}(i,j)$, the error estimate at level l. Update the global error E using $E \leftarrow 2E + E_l$.

5) Repeat step 4 three more times, letting l take on the values 2, 1, and 0. At this point the global error vector E is the sum of the errors computed at each level, weighted by the scale at each level: $E = 8E_3 + 4E_2 + 2E_1 + E_0$. Return this value as the LLT estimate of the tracking error.

The high-level tracker: Where the low-level tracker follows targets by iterated template matching, the high-level tracker does so by searching the neighborhood of the estimated target position for a match to an explicit model of a human face. Several face matching algorithms have been used, the most robust being an implementation of the eigenface method of Turk and Pentland [55]. Since this method is similar to SLAM, however, we instead describe a faster method based on feature matching.

The feature matching version of the high-level tracker expects faces to consist of a pair of roughly symmetric curves (the sides of the face) surrounding a cluster of small high-frequency features (the eyes, nose and mouth). It extracts features using a series of convolutions and heuristic tests, and then searches for a set of image features that match the model within some tolerance. If it finds an acceptable match it halts and returns the position of the model center, otherwise it returns failure. The search computation is essentially a graph matching process like that used in the Image Understanding Benchmark, but is expressed procedurally.

The specific steps of the computation are as follows:

1) convolve the image with 8x8 edge templates extracted from the sides of actual faces, and apply a threshold to obtain a binary image.

2) label connected components in the binary image and compute area, bounding box et cetera.

3) apply heuristic tests to pairs of components based on area, horizontal separation, vertical alignment, and aspect ratio. The result is a list of pairs of curves that are plausible edges of faces.

4) convolve the image with a 7x7 LOG mask and extract zero crossings

5) label connected components in the zero crossing image

6) apply heuristic tests to sets of zero crossing components to eliminate those that are too large or are otherwise implausible as facial features

7) check to see if any accepted facial feature sets are appropriately positioned within a pair of accepted face edges. If so report success, and return the location of the center of the circle defined by the two face edge curves. Otherwise report failure.

The control process: The task of the control process is conceptually very simple. The low level tracking processes always returns a position estimate in the form of an error vector, which is the difference between the center of the target pyramid in the current image and the best match in the search pyramid. The control process simply adds this error vector into the running estimate. The high level tracker may return the absolute location of a face in the image, or it may not return anything at all. If the high level tracker returns a value, the control process records it as the current target position estimate.

3 Available Software Technology

The first two sections of this report dealt with AVIS user needs, describing the kinds of computations they need to perform, the programming paradigms they rely on and the environments in which their work is done. In this section we examine the state of the art in several key software technologies that appear critical to successful use of AVIS accelerators. There are two reasons for doing this. The first is simply to provide guidance to AVIS development groups. The second is that the current state of the art defines what AVIS users are likely to expect in the way of software support. If AVIS accelerators are delivered without comparable software tools, users will be disappointed and will resist using them.

We have assumed in what follows that the AVIS program will not attempt to leapfrog the state of the art in software technology, because of the development costs and technological risks that would be involved. Thus we have restricted the survey to approaches that have either demonstrated success on significant applications, or have substantial commercial and institutional momentum and are clearly technically feasible. We have omitted speculative approaches based on non-traditional or unproven programming paradigms.

The discussion is divided into subsections following the taxonomy of software types outlined in section one. The first subsection reviews development and end user environments, which provide the services used by software developers to assemble complete applications. These environments may include application-specific libraries, GUI-based application builders et cetera. The second subsection deals with lower level tools such as compilers and debuggers, and the third discusses operating system interface and control software.

3.1 Development and User Environments

The majority of users in the domains covered by this study are involved in research and development of some sort. They spend most of their time devising new ways to perform some image understanding task, or constructing new systems to address a specific image understanding problem. Thus their primary interaction with AVIS systems will occur in the course of building and testing a either a new computational primitive or a complete system. In order to use the accelerators effectively, they need software tools that allow them to perform these activities with a minimum of effort.

In this section we review tools that support assembling systems from pre-existing components. This activity may be sufficient to allow prototyping an application or a new computational primitive, if the set of available components is rich enough. If new components must be created, the user may need to resort to lower level tools. Experience has shown that good development environments are still very useful however, because they make it easy to test and evaluate the new component once it is built. They also make it easy to reuse the component in hypothetical future applications.

Development environments for image processing have a long history [28] and may provide take on a variety of forms, depending on the type and level of service they provide. They generally perform one or more of the following functions:

- **encapsulating primitives.** Most image processing environments provide encapsulated implementations of standard image processing operators such as convolution, point transforms et cetera. The primitives may be presented via subroutine libraries, as stand-alone executable programs, or as primitive commands in a textual or graphic shell.
- **analysis and display.** Most systems provide some facility for examining the results of image computations, e.g. displaying images on a monitor. Some systems include sophisticated data analysis and plotting tools.
- **interactive composition of primitives.** In addition to the above, most modern systems provide the ability to invoke sequences of primitive operations interactively and observe the results.
- **software engineering.** The largest and most complex systems provide support for large development efforts, and include facilities for version control, image and code database management, and automatic interface generation.

In the remainder of this section we review several state-of-the-art environments that are either in widespread use or appear likely to succeed because of widespread support among researchers. The list is not exhaustive but is intended to cover the range of types of systems that are reasonable candidates for AVIS delivery environments. Along with each environment we include a discussion of features affecting its suitability for the AVIS program and, where appropriate, an outline of how it might support some of the sample applications identified in the domain study of section 2.

3.1.1 The Image Understanding Environment (IUE)

The Image Understanding Environment (IUE) is a massive object-oriented software system designed to provide a common substrate for image understanding research and development. It will consist primarily of a C++ class library of image objects and geometric primitives, with associated algorithms for their manipulation. A user interface and data exchange tools will be built on top of these core objects. IUE development being managed by Amerinex Artificial Intelligence, Inc., with ADS, Carnegie-Mellon University, and Colorado State University as major subcontractors. The system is currently scheduled to be completed in 1996. Current documentation and status information can be obtained from Amerinex at `http://www.aai.com`.

The fundamental goal of the IUE program is to improve the productivity of IU researchers by allowing them to share and/or reuse code. It also encourages a standard view of what the major conceptual issues in the design of an IU system are. This shared vocabulary will provide secondary benefits to education, and will encourage transfer of IU technology to industry. At present the IUE is not designed with real-time system building in mind. However, the object-oriented nature of the system should facilitate use of special hardware, and a group of European researchers is investigating issues raised by real-time applications.

The main components of the IUE are the class hierarchy, the data exchange facility and the task facility.

The IUE Class Hierarchy

The IUE class hierarchy is the major intellectual contribution of the IUE to image understanding research. It is intended to capture most of the important entities in the IU problem: objects, sensors, and the image formation process. It is dauntingly complex, with over five hun-

dred distinct object types, and it is unclear whether the average researcher will be able to make use of its power. The design appears to be very sound, however, and IU researchers seem eager to make use of the system and contribute to its success.

The class hierarchy is documented in an evolving document available from Amerinex [34]. The most important functional classes of the system are:

- **Spatial objects.** Conceptually, an IUE spatial object is a point set in an n-dimensional space. Spatial objects are typically used to represent world objects being imaged. However, they are extremely general and can in principle represent abstract mathematical concepts such as manifolds, hyperspheres, or other useful objects. Important subclasses of the general spatial object include points, curves, surfaces, and volumes. Spatial objects can be constructed from other objects by set operations in the manner of constructive solid geometry. Methods are provided to find the nearest point on an object to a probe point, and to find the surface normal at a point.
- **Images.** IUE images are extremely general; they can include multi-band, volume, and/or time series data, and can be grouped into complex structures such as pyramids, mosaics and so on.
- **Image features.** The IUE feature class is intended to capture the data structures needed for intermediate-level vision. Typical examples might include line segments, regions, or parametric curves. Image features are a subclass of spatial object and inherit all of the methods that apply to the parent class. In addition, they store information that describes how they relate to the image from which they were extracted.
- **Sensors.** The IUE distinguishes between sensors considered as physical devices and sensors considered as sources of data. The physical sensor is considered to be a function that maps a portion of some space containing spatial objects and attributes to a space containing values, i.e. an image. In its role as data source, the sensor is viewed as a function of location (i.e. image coordinates) that returns a value, combined with attributes that describe the state of the sensor when the image was formed.
- **Coordinate systems.** Most IUE objects are defined in terms of a coordinate system. The IUE provides classes for defining various types of cartesian and polar coordinate systems and for mapping objects from one coordinate system to another.

The IUE Data Exchange Facility

The IUE defines a standard format for exchanging IUE objects between dissimilar hosts, storing them in files and so on. The format relies on a LISP-like syntax expressed in ASCII characters, so it is extremely portable. Binary data is not permitted, so IUE DEX files are not suitable for storing image data except for transport. DEX files can contain references to other files, however, so it is relatively easy to construct local formats that use efficient binary storage for pixel data, and human-readable ASCII storage for structure definitions and higher-level objects.

The IUE Task Facility

The IUE is unusual in that it allows computations themselves to be objects, rather than concealing all computations within methods of the data structures they apply to. The primary mechanism for this is the Task object class, which encapsulates a single large-grain image processing computation. A specialization of the Task is the Data-Generator class, which can serve as

a source of data for the computation, and provides for iteration and looping in the dataflow. Collections of Tasks can be interconnected to form a Dataflow-Graph, which can specify a complex computation as a sequence or network of Task operations. Once connected into a Dataflow-Graph, a set of tasks can be collapsed into a single Compound-Task, which can itself be inserted into another Dataflow-Graph.

The class definition for each Task object defines the interface to the computation, such as its required inputs and the outputs it produces. It thereby provides the information necessary to insert the task into a dataflow graph, where the outputs of one task can form the inputs for others. The inputs and outputs are of the Task-Parameter class, and can take on any value. Tasks also have attributes to facilitate user interaction with them, such as a documentation string, a list of keywords related to the task's semantics, and an icon for use in a graphical user interface. The status and history of a computation can be tracked via the use of the task's Process-Status attribute and by examining the explicitly represented parameters.

The actual implementation of a Task's algorithm is not contained within the Task itself, but rather in its Process-Specification attribute. The process specification contains the executable code for the algorithm, which may be in Lisp or C/C++, may be a dataflow graph itself, or may even be an external process that can potentially be executed on a separate processor. Although the IUE itself will not provide parallel execution of tasks, the Process-Specification mechanism has the potential to allow users to define parallel implementations themselves. The parallelism can be internal to the task (by implementing the algorithm on a special- purpose accelerated processor), or be external to the task (by running multiple tasks in parallel on separate processors). The separation of the interface (the Task class) from the implementation (the Process-Specification class), enables the definition of different implementations of a given task, depending on the available hardware.

Related tasks can be organized into Task-Groups. A proposed list of IUE defined Task-Groups includes

- Display, browsing, and editing tools that allow the user to explore IUE objects or the IUE class hierarchy.
- Image processing routines that perform low-level image computation
- Segmentation and grouping operators for intermediate-level computations
- Model fitting tasks that fit parametric models to data
- Matching tasks that perform search to find mappings between collections of image features and object models
- Modelers that aid the user in constructing object descriptions for testing or modelling purposes.

Users will be able to define additional task groups as necessary.

Image Operators

A major question that remains open as of this writing is what low-level image processing operations will be available in the Image Processing task group or as methods of IUE Image objects. The original plan was to implement the Programmers Imaging Kernel standard (PIKS, see below), but there are problems with the PIKS data format and with the amount of work required to

implement the PIKS standard. The possibility of adopting the low-level methods from Khoros (see above) is being explored.

For intermediate-level computations, table 3 shows a set of algorithms that have been pro-

Table 1: Proposed intermediate-level primitives for IUE

task	task implementations
Edge detection and linking	Roberts, Hueckel, Sobel, Nevatia-Babu, Burns, Canny
Corner detection	Binford-Horn, Kitchen, Moravec, Plesey, Noble
Region growing	Brice-Fennema, Price-Ohlander, Feldman-Yakimovsky, Horowitz-Pavlidis, Hanson-Riseman
Curve/segment construction	Roberts, iterative endpoint, Ramer, Horowitz-Pavlidis, Hough, Brady-Asada, B-spline, Bookstein, Porril

posed for inclusion as Tasks. This portion of the IUE specification is particularly unstable, however, so this list may change radically in the final implementation.

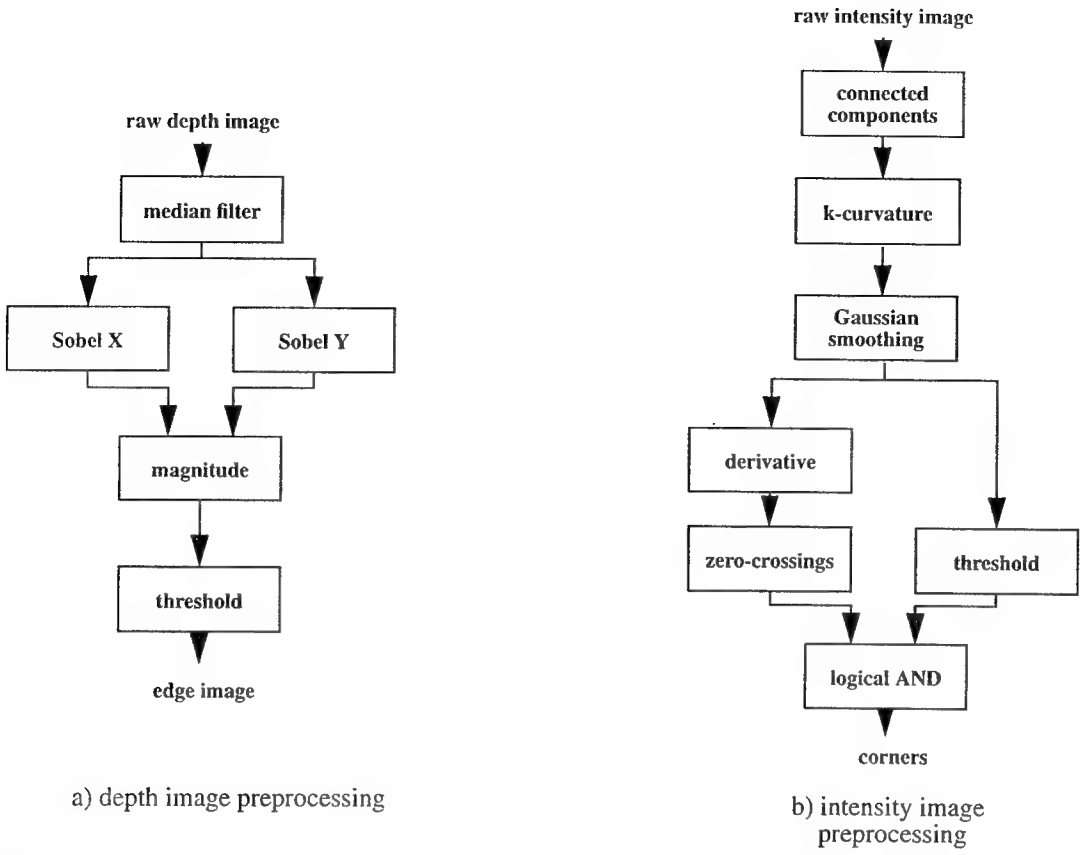
IUE and the sample tasks

The IUE is by far the most ambitious of the software environments discussed in this report. Thus it is not surprising that it offers quite good support for the sample tasks identified in section 2. It should be noted that the IUE has not actually been implemented, however, so discussions of its usefulness are necessarily speculative. Here we outline strategies for implementing each of the sample tasks using IUE tools, and discuss how well the IUE supports each task. The discussion is based on the IUE Class Definitions (IUE-CD) document [34], version of October 1994.

Image Understanding Benchmark (IUB). The IUB tasks includes subtasks in early vision, intermediate vision, and high-level vision. All of these are addressed by the IUE class library. The low-level operations are suited for implementation as IUE Tasks (IUE-CD Chapter 15). The intermediate level operations can be based on IUE Image Feature methods (IUE-CD Chapter 5). The high-level processing can be done by instantiating IUE Spatial Objects (IUE-CD Chapter 4) and writing the necessary C++ code to reason about these spatial objects.

In the IUB, initial processing is done over both a depth image and an intensity image. Both of these images can be instantiated as IUE-2d-scalar-images (IUE-CD p.93). Using the IUE task mechanism, the depth image will be fed through a median filter, and then a gradient magnitude image is created by convolving the median filtered image with Sobel kernels in both the X and Y directions. The gradient magnitude image is then thresholded to obtain an edge image. The IUE_dataflow_graph (IUE-CD p.554) that produces the edge image from the raw depth image is depicted in figure 2a).

The initial processing on the intensity image consists of labeling the connected components in the image, and computing the k-curvature of the component boundaries. Labeling the connected components can be done by an IUE Task implementing a 4-connected component algorithm. It it not yet known whether the IUE will provide such a task, or whether the IUB implementors will



a) depth image preprocessing

b) intensity image preprocessing

FIGURE 2. IUE task networks for Image Understanding Benchmark preprocessing

need to write it themselves. The regions themselves can be represented as IUE_2d_image_regions (IUE-CD p.370). The boundaries of the components are processed to determine the k-curvature at each boundary point by using a table look-up to estimate the angle that the boundary forms at that point. The IUE_2d_image_region class has methods that return both the inner and outer boundaries of an image region, which can be used in this task. The raw k-curvature values are then smoothed by a 1x7 Gaussian mask. Then one takes the first derivative of the result by convolving with a mask of {-1,1}, and zero-crossings in the derivatives are found. Corners are taken to be zero-crossing points whose smoothed curvature exceeds a given threshold. The IUE_dataflow_graph implementing these operations is depicted in figure 2b. The output of this graph is a set of corners, which are objects subclassed from an IUE image feature class such as the IUE_image_vertex class (IUE-CD p.378). Note that IUE tasks are not restricted to having images as their inputs and outputs.

The IUB's intermediate level operations use the corners found above to determine the locations of rectangles in the image. The Graham scan algorithm is to be used on the corner point sets, and additional processing is done to see if the corners can be grouped into a rectangle. The IUE does not have direct support for these algorithms, which would need to be implemented as user code. The rectangles can be instantiated as IUE_2d_rectangles (IUE-CD p.302). Once the rectangle is found, its intensity and other statistics are extracted from the image. A subgraph match between the resulting set of rectangles and a set of candidate models is performed. The IUE does not support subgraph matching directly, and this step must be implemented by user code.

Confirmation of the model selected is performed using a top down search incorporating a spatially local Hough-transforms. Using the hypothesized rectangle locations, a region of the depth image is selected for application of a hough transform for lines. The edge points found in the region of the depth image can be added to an IUE_2d_pointset (IUE-CD p.268), which has a hough_transform_lines method that returns a set of lines. These lines can be checked against the hypothesized model and the results are then printed.

It is still unclear to what the extent the still evolving IUE will implement the IUB algorithms. However, the necessary data structures enabling one to implement the IUB within the IUE framework will be available.

Software Library for Appearance Matching (SLAM). The IUE class library will not provide direct support for the high-dimensionality space manipulations needed for SLAM. Most of the IUE spatial object classes have a single abstract superclass, which cannot instantiate members directly. The IUE will then implement two instantiable subclasses, a two-dimensional version, and a three-dimensional version. An example of this is the IUE_vector class (IUE-CD p.261), which is abstract, and the classes IUE_2d_vector and IUE_3D_vector, which are instantiable. Most of SLAM deals with arbitrary dimensional spaces. Although the IUE_vector class can be subclassed, most operations on SLAM vectors will not be supported directly by the IUE. Hence, using the IUE would require that most of the SLAM application be written as C++ code. However, the IUE model is very general, so the SLAM application can be written with the IUE framework.

The first step in the SLAM application is to prepare an image database as feature vectors. The images can be instantiated as members of the IUE class IUE_2d_scalar_image (IUE-CD p.93). Each image is preprocessed if necessary (probably via IUE Tasks, IUE-CD Chapt.15), and then converted into a vector for further processing. The IUE has an abstract class IUE_vector (IUE-CD p.261) which can be subclassed to provide the data structure needed to contain the feature vectors. We will call this arbitrary length vector subclass the SLAM_vector. The SLAM_vector should have a constructor that takes as input a IUE_2d_scalar_image and produces the corresponding vector.

Next one would use the collection of SLAM_vectors to construct the eigenspace. The eigenspace can be thought of as a coordinate system, so it is appropriate to use the IUE coordinate system object (specifically, the IUE_nd_cartesian_coordinate_system class (IUE-CD p.487)) as a superclass for this eigenspace. The current version of the IUE-CD document does not specify whether this class is abstract or instantiable, and only specifies one attribute, an integer representing the dimensionality of the coordinate system. We assume that one could create a SLAM_coordinate_system as a subclass of IUE_nd_cartesian_coordinate_system. One constructor for the SLAM_coordinate_system class would take as input a collection of feature vectors, and an indication of how much to truncate the basis of the eigenspace to form the SLAM_coordinate_system.

Once the SLAM_coordinate_system object is instantiated, feature vectors can be projected onto it, using a transform() method of the SLAM_coordinate_system class. Sequences of projected points can be joined to form B-spline interpolated manifolds. These are n-dimensional manifolds in m-space, with $m > n$. The IUE object hierarchy only has one- and two-dimensional manifolds in two- and three-space. Examples of these are curves (1-D manifolds, IUE-CD p.270) in 2 and 3-space, and surfaces (2-D manifolds, IUE-CD p.307) in 3-space. Curves and surfaces both have instantiable b-spline IUE subclasses, but the fixed dimensionality of these objects

makes them inapplicable to SLAM. Creating arbitrary dimensional manifolds within the IUE requires subclassing the superclass of curve and surface, i.e., the abstract class IUE_spatial_object (IUE-CD p.232) to create a SLAM_bspline class. This class contains the attribute coordsys, which in our case would be a pointer to the SLAM_coordinate_system constructed in the previous step. The IUE_spatial_object class contains several methods which are useful for SLAM, but are of course unimplemented, since the IUE_spatial_object class is abstract. Of particular interest is the nearest_point method, which accepts a point as input, and returns the point on the spatial object closest to the input point, and the nearest_distance method which returns the distance to the nearest point to the input point.

Given the classes described above, an IUE version of the SLAM application written in pseudo C++ might look as follows:

```
// Learning step

SLAM_coordinate_system eigenspace[MODEL_NUM];// eigenspaces for each object
SLAM_bspline manifold[MODEL_NUM];          // the manifolds for each object

  for (i = 0; i < MODEL_NUM; i++) {

    // Convert the images to feature vectors
    list<SLAM_vector> full_dimensional_vect_seq;
    for (all IUE_2d_scalar_images IMG of model[i])
      full_dimensional_vect_seq->insert(SLAM_vector(IMG));

    // Construct the eigenspace
    eigenspace[i] = SLAM_coordinate_system
      (full_dimensional_vect_seq, truncation_amount);

    // Project images into eigenspace
    list<SLAM_vector> reduced_dimensional_vect_seq;
    for (all vectors V in full_dimensional_vect_seq)
      reduced_dimensional_vect_seq->insert(eigenspace.transform(V));

    // Construct the manifold
    manifold[i] = SLAM_bspline(reduced_dimensional_vect_seq);
  }

// Recognition step

int nearest_model;                // index of the nearest model
SLAM_vector nearest_point;        // location of nearest point on nearest model
double min_distance;              // distance to nearest point on nearest model

for (all images IMG to be recognized) {
   min_distance = MAX_DOUBLE;                // initialize min_distance
   SLAM_vector feature_vect(IMG);            // construct feature vector
   for (i = 0; i < MODEL_NUM; i++) {         // for all models:

      // project the feature vector into the eigenspace
      SLAM_vector projected_vect = eigenspace[i]->transform(feature_vect);
```

```
        // find the distance to this model
        double distance = manifold[i].nearest_distance(projected_vect);

        // Save the best match so far
        if (distance < min_distance) {
           min_distance = distance;
           nearest_model = i;
           nearest_point = manifold[nearest_model].nearest_point(projected_vect);
        }
     }
     // output best match
     print_answer(nearest_model, nearest_point, min_distance);
}
```

A good implementation of SLAM using the IUE would be generally useful to the IUE community, since it would require implementing n-dimensional versions of the abstract classes vector, curve, etc.

Hierarchical Tracker

As is the case with the other sample applications, the IUE has data structures that can be used in the tracking application, but most of the algorithms will need to be coded by the application programmer. The tracker is a two-stage algorithm, with low and high-level processes. The low level performs a coarse-to-fine search in a Gaussian pyramid. The IUE has an abstract class IUE_pyramid_image (IUE-CD p.88) that can be subclassed to serve as the data structure that holds the Gaussian pyramid. Constructing the pyramid itself will probably be a method of the IUE image class. If not it can be accomplished via a IUE Task graph consisting of repeated Gaussian convolution followed by subsampling. The IUE_pyramid_image class supports get and set operations on pixels and windows, through which the remainder of the low-level tracking algorithm (SSD, bilinear interpolation, etc.) can be implemented. It is also likely that the IUE image class will contain operators to perform these operations directly; however, the IUE image operator set has not been defined as of this writing.

The high level algorithm is a feature-based approach based on matching image features against a model. The image features can be represented using the IUE image feature data types (IUE-CD Chapter 5). Extracting the features requires convolutions and connected component algorithms that can be done as IUE Tasks. The output of the connected component algorithm can be instantiated as IUE_2d_image_regions (IUE-CD p. 370). The IUE_2d_image_regions can be filtered on the necessary criteria. The high level process looks for appropriate spatial relationships between image features. These relationships can be instantiated as IUE_2d_perceptual_groups (IUE-CD p.374), which maintain such information as the orientation of the group and its bounding box.

The methods described above will allow the tracking computations to be performed using IUE data structures and methods. However, the IUE is explicitly not intended for real-time applications, and hence has no facilities for expressing the idea that the two tracking algorithms prescribed in the task description are to be executed concurrently, or for bounding their running time. An IUE-based implementation of the tracker would have to rely on operating system facilities for these aspects of the task. It is also worth noting that the IUE objects and methods are designed to favor generality and reusability over execution speed. Thus they may impose a performance penalty that is unacceptable for real-time tasks. Given the relative immaturity of the

IUE, it is difficult to determine the extent to which this will be a problem, or to determine whether the task computation will be supported directly by IUE methods. However, it is clear that the necessary data structures needed for the computation will be available.

3.1.2 Khoros

Khoros [41,42,24] is an environment for developing image and signal processing applications, and for examining and visualizing data. It was developed at the University of New Mexico under Prof. John Rasure, and is now distributed by Khoral Research Incorporated. It is available under a variety of licensing plans that vary in cost and in the rights that they confer; however, a basic license to use the software is free.

Khoros consists of a complex hierarchy of software tools and libraries. It provides facilities for rapid prototyping of image processing applications, for displaying images or plotting data, and for creating new image processing algorithms and integrating them into the Khoros environment. It also provides support for building distributed interactive tools (groupware), and CASE facilities such as automatic generation of graphical user interfaces and documentation.

Khoros has achieved tremendous success in several communities of interest to the AVIS program. In particular, it has been adopted by many ATR groups, and is now required in many ATR contracts. In part because of this, Khoros continues to evolve rapidly. As of this writing KRI is in the early stages of releasing Khoros 2.0, which is a redesign and reimplementation of almost the entire system. The discussion presented here is based on Khoros 2.0. We caution that the documentation for 2.0 is incomplete and that some of the observations made here are based on our inferences about how the final system will work. More up-to-date information is available from the Khoros Web site, `http://www.khoros.unm.edu`, and via the internet newsgroup `comp.soft-sys.khoros`.

Existing Khoros documentation proposes a variety of ways of partitioning the system into components. In our view the primary features of the system are:

- **The Cantata visual language.** Cantata is a sophisticated graphical shell that allows users to construct complex networks of image processing programs interactively. It also serves as a launch point for other Khoros tools
- **The program library.** The programs that Cantata users manipulate are drawn from a huge library of programs for image processing, signal processing and matrix computation.
- **Display and visualization tools.** Results of Khoros computations can be viewed using a variety of image display, image editing, and 2D or 3D plotting tools.
- **Code generators.** The Khoros library can be extended with user-supplied programs that are kept in user-controlled file storage. Khoros provides several tools for generating the various hooks, command-line options and descriptor files that are needed to make the programs accessible within Cantata. The tools include programs for generating graphical user interfaces that are compatible with the look and feel of the rest of Khoros.
- **Programming services.** Programmers writing under Khoros have access to libraries of low-level routines that support structured I/O and that provide host-independent access to operating system services. In most cases they can also call library routines from C code that implement the functionality of the Khoros program library.

Cantata

Cantata is by far the best known and most visible component of the Khoros system. It is a visual programming language based on coarse-grained dataflow, with extensions for iteration, conditional execution and procedural abstraction.

The Cantata interface contains a workspace canvas and a top-level menu. A Cantata program is defined by a directed graph, where each node represents an operator (a library function) and each arc represents the flow of the data. After a given operator processes its data, the result is passed out along its output arc to the next node for processing. Execution continues until all of the operators have been fired. Cantata can instantiate any of the library functions into its workspace, although not all of them are included in the top-level menu.

Each node is initially displayed on the workspace as a block with various icons on it. At this point, the operator is in its *glyph* state. A glyph is the iconic representation of an operator. Because operators may have a large amount of information associated with them (e.g. control parameters), they may be displayed as *forms* at any time by clicking on one of the glyph icons. Forms contain all the detailed information about a given operator in an editable pop-up worksheet. Using glyphs yields a more compact representation of the application, yet forms are necessary to edit operator information. The glyph/form representation addresses the need for both simplicity in the graphical representation and detailed information for each of the routines. It provides an effective method for maintaining a lot of information in a small space and accessing it relatively quickly.

In Khoros versions prior to 2.0, the decision about when to execute the programs making up a Cantata graph was made by a scheduler built into Cantata. When the user enabled execution of the graph, the scheduler traversed the list of ready nodes, spawning processes by local or remote fork/exec. The scheduler was responsible for insuring that no node ran before its inputs were available, and for interpreting the action of control nodes such as IF and LOOP. In version 2.0 the mechanism has been changed to move decision-making closer to the individual nodes. This is intended to facilitate distributed execution of Cantata graphs. As of this writing, flow of control operators are not supported in version 2.0.

The Program Library

Part of the motivation for Khoros was to encourage creation of a set of reusable programs for common image processing tasks. It has succeeded to such an extent that Khoros now partitions its libraries into 'toolboxes' of related programs. This allows code that is specialized for e.g. GIS applications to be kept separate from more general-purpose programs. The complete list of routines is too large to reproduce here, but includes all of the standard image processing primitives (see discussion of PIKS below) as well as matrix operations and test image generation routines.

Routines in the program library consist of stand-alone UNIX executables that follow particular input/output conventions. In general the program consists of a main program that interprets the arguments and then calls a library routine to do the actual computation. In most cases the library routine can also be called from any C program. In our conversations with Khoros users we found some who never use Cantata, preferring to access the system exclusively via the library API.

Display and Visualization tools

Khoros supports interactive exploration of image and matrix data via the *editimage* image editor and the *xprism2* and *xprism3* plotting tools. Editimage accepts image files in Khoros

Visualization/Image File Format (VIFF) format. It can display images using a variety of color tables, and allows the color table to be modified interactively. It also provides display functions such as panning over large images, zooming, and thresholding.

Xprism2 is a two dimensional plotting package which features 2D, discrete, bar, polymeric, and linemarker plots. There are several acceptable forms of input data: image files, files with functions or data points, and functions entered from the keyboard. The data may be entered as (x,y) pairs or just y points. Xprism2 provides an interface for scaling, rotating, and translating graphs. It also allows the user to change fonts, colors, axes, marker, line, and plot types, and titles. Plots may be saved in Image File Format and output may be sent to various printer types.

Xprism3 is a three dimensional plotting tool which features 3D, scatter, impulse, mesh, horizon, surface, contour and color mesh plots. The input data forms are the same as Xprism2. Xprism3 accepts (x, y, z) triplets or just z data. It allows the same modifications as Xprism2 through a GUI. In addition, it provides a way to change the perspective along the three axes. This feature is impressive because of its speed.

Code Generators

Khoros provides several programs to help automate the task of creating a Khoros-compatible application and putting it into the Khoros library (or more properly, into a toolbox). Such programs may make use of Khoros facilities to provide a graphical user interface with menus, buttons et cetera. A GUI created in this way can be laid out interactively using yet another tool. Although each tool is a stand-alone program, they are typically launched from within Cantata or another tool.

The code generators provided with Khoros include:

- **craftsman** - a tool for managing toolboxes. Craftsman allows the programmer to create, copy and destroy toolboxes, and to invoke Composer on their contents.
- **composer** - a tool for editing toolbox contents. Composer's primary task is to insure that the various files associated with a Khoros program (including source code, documentation, and user interface specification) are kept consistent and are stored in the right places. It also checks library dependencies in the manner of Unix `make`.
- **guise** - a tool for designing Khoros-compatible GUIs. Guise allows the programmer to place graphical widgets on a pad, lay them out as desired, and connect their functionality to the parameters of a Khoros library program.

Programming services

Khoros provides abstraction layers that conceal basic operating system services behind a system-independent API. This eases porting Khoros applications to new architectures. Version 2.0 extends the idea by providing a Khoros-aware I/O subsystem that allows user programs to read or write Khoros objects without having to understand the associated data types. Among other things, the I/O system allows programs to communicate via temporary files, shared memory objects, or sockets without any explicit knowledge of the communication mechanism being used. It also allows automatic coercion of data into the type and size needed for the application. For example, a program that computes Fourier transforms of images can assume that its input is an image of type Complex, and that the row and column dimension is a power of two. The I/O system will auto-

matically coerce the pixel type to complex during loading, and will resize the image by padding or interpolation if needed.

Khoros as a development environment

Because Khoros is part of the working environment of many potential AVIS users, we have been at pains to work with the system and to ask current users what facilities they find especially useful. The most commonly cited virtue of the system is the large library of image processing primitives. The fact that implementations of most standard operators a) exist and b) are cleanly interoperable is tremendously useful. The second most popular feature of Khoros is the Cantata visual programming environment. Users find Cantata very effective as a means of assembling Khoros and user programs into larger applications.

The Khoros users with whom we spoke were less enthusiastic about the code generation tools and programming support. They felt that the learning curve was excessively long, that the large number of tools was confusing, and that the functions of many of the tools overlaps or is unclear.

It is perhaps unfair to criticize the documentation for Khoros 2.0, since the system is early in its release cycle and will doubtless improve in later releases. At present, however, the printed documents are remarkably frustrating to use. There are over two thousand pages of documents in the current release, but the documents are unindexed and highly repetitive, and their organization is unclear. In general they consist of functional descriptions of pieces of software, and are of little use in answering questions of the form "how do I do X?". There is a desperate need for more code examples and tutorials that will help users find the information they need.

Khoros and the sample tasks

Like most of the systems reviewed in this section, Khoros is fundamentally oriented toward low-level image processing. Thus it only contains primitives for parts of the sample tasks described in section 2. In evaluating Khoros (and the other systems) as implementation environments for these tasks, the important questions would seem to be

- what portions of the task are supported?
- how much work is involved in expressing the supported subtasks? and
- how will any new code that is required fit into the computational model(s) that the system provides?

There are three basic approaches that can be taken to implement the sample tasks under Khoros. The first is to work within Cantata as much as possible, and to write additional Cantata-callable routines to handle the unsupported parts of the task. The problem likely to arise here is that although Cantata does provide some control flow capability, it is far less flexible than a conventional procedural language. The second option is to work in C or C++, making use of Khoros services via the Programming Services API and implementing the required new code as Khoros-compatible library routines. This approach gives up the advantages of an interactive rapid prototyping environment, but provides much more flexible flow of control. Problems may still arise due to the lack of a rich set of high-level data structures in Khoros. The third option is to implement the new code in more or less raw C/C++. This solves the data structure problem but is least likely to result in code that can be reused for other applications.

It is unfortunate that there is apparently no way to combine straight-line C code with image computations defined using Cantata. There is a facility for converting a Cantata workspace to an

encapsulated form which can be run from the UNIX command line, but it does not appear possible to call a Cantata workspace from an arbitrary C program. This means that programmers must choose between the interactivity of Cantata and the flexibility of C code. They cannot have both in the same program.

The Image Understanding Benchmark

Khoros provides relatively little direct support for the IUB task. For the depth image preprocessing step (see figure 2), the `vqmed`, `igradient` and `kthreshabove` operators can perform the required median filter, Sobel gradient estimator and gradient threshold on floating point data. For the intensity image, connected component labelling can be performed using `vlabel`; note however that `vlabel` is intended to use a very general notion of region uniformity, and that it may be very inefficient for the IUB task. In addition, it will not produce exactly the same set of integer labels as the algorithm described in the benchmark.

The rest of the preprocessing and hypothesis formation subtasks will require new code. If the implementation plan involves breaking the tasks down into multiple Khoros-compatible routines, the Khoros data services will provide some support for passing geometric primitives (e.g. lists of vertices or convex hulls) between different sections of the program. In particular, the Khoros Geometry model (see Data Services in the Khoros documentation) would provide a convenient way to represent object models, chain codes, and convex hulls.

The higher level portions of the task mostly involve graph manipulations and will need to be written from scratch, using data structures defined by the programmer. However, some of the higher level steps involve probing the image to test the reasonableness of hypotheses, and these operations can benefit from Khoros routines. The first probing step searches a window around the expected location of a rectangle in the depth image, and counts pixels that are too near, far enough and too far to be part of the rectangle being sought. The counts form the basis of a match strength for the rectangle. The windows can be extracted with `vextract`, `kclip` can be used to classify pixels into the desired classes, and `kstats` can be used to count the pixels. The second probing step involves performing a Hough transform on local regions of the thresholded depth gradient image. Khoros provides no direct support for this step, but adding a Hough transform routine to the library would obviously be useful.

SLAM

The SLAM task involves considerably more image manipulation than is required by the IUB. Consequently, Khoros does a much better job of supporting it. SLAM offers the option of preprocessing training or input images using a variety of normalization, smoothing, derivative and transform operators. All of the desired operators are available as Khoros primitives. The SLAM GUI could be implemented fairly straightforwardly using the Khoros GUI-building tools.

In addition to its image and geometry operators, Khoros provides a small library of numeric routines. These can be used to implement the training step of the algorithm. The average image could be formed and subtracted from the training image using pixelwise arithmetic operators. A new routine would need to be written to construct the training vectors from the input images. The basis set could then be determined using `mmul` (matrix multiply) and `meigen` (eigenvectors and values). Note however that `meigen` is a very general routine, and that code that takes advantage of the symmetry of the training matrix would be much faster. A routine would have to be written

to extract the k eigenvectors with maximal eigenvalues. `Mmul` could then be used to project the training images onto the set of eigenvectors.

Khoros provides no explicit support for building the manifolds representing the projections of each training set on the eigenspace(s). The Geometry data types could be used to represent the resampled manifolds, however. New routines would have to be written to perform the matching step in which the projections of new images are compared to each manifold.

Hierarchical tracker

Implementing the hierarchical tracker under Khoros raises efficiency issues that were not as important in the other two tasks. In earlier versions of Khoros each operator was explicitly identified with a distinct UNIX process, and all communications were mediated by the UNIX file system. The evolution toward version 2.0 has relaxed this, so that communications are now abstract and can be implemented by (e.g.) shared memory objects. Khoros operators are now less clearly tied to processes, and can in any case be called from C code in their library versions. Still, the default assumption in the design of Khoros has been that the operators are large, heavy-weight constructs, and that the data objects they manipulate are large. These assumptions have led the Khoros designers to stress abstraction, for example by arranging for Khoros data objects to be accessed by what amount to database operations. This helps users write code that works for a wide variety of inputs; however, it imposes overhead that may be disastrous for time-critical applications that manipulate small images or regions of interest. In the discussion that follows we ignore these issues, but caution that they may preclude using Khoros for any real-time application.

The tracker is divided into separate low-level and high-level tasks, which run asynchronously. Although the mechanism is not yet clear, it appears that Khoros 2.0 will be able to support asynchronous execution. The low-level task begins by constructing a Gaussian pyramid. This can be done in Khoros by using vconvolve and kshrink to repeatedly blur and subsample the image. Separable convolutions can be used to make the computation efficient. The next step of the low-level task is to extract target and search pyramids from the current and new image pyramids, using bilinear interpolation for subpixel accuracy. Eventually the Khoros `kresample` routine will be able to do this, but at present it does not support interpolation. Finally, the low-level task performs coarse-to-fine search for the offset between target and search pyramids that minimizes the pixel-wise sum of squared differences. SSD computation could theoretically be done using `vextract` to extract windows, `karith2` to subtract them, `ksqr` to square the result, and `kstats` to compute the sum of squares. In practice this would be hopelessly inefficient, and a new routine will be needed.

The high-level tracking task looks for groups of features that match its model of a face. The task begins with several convolutions, which can be performed using `vconvolve`. Zero crossing detection can be performed by a combination of convolution and thresholding, or a new routine can be written. The remainder of the high-level task consists of connected component labelling and search, and would have to be done in user code.

3.1.3 The Programmer's Imaging Kernel System

The Programmer's Imaging Kernel System (PIKS) is an API (applications programming interface) and a set of associated data types for computing on images and image-derived data structures. It provides operators for image analysis, manipulation, enhancement, and filtering,

plus a small number of functions for pixel classification and grouping. It includes most of the operators used in low-level computer vision computations, and hence is a good source of information about what sort of operations AVIS users need to perform.

PIKS is part of an emerging ISO standard called the Common Architecture for Imaging (CAI). The full CAI standard is divided into three volumes. The first (ISO/IEC 10287-1) defines an abstract model of image computing and a set of data structures for representing images and image-related information. The second volume (ISO/IEC 10287-2) defines the PIKS API. The definition is in terms of abstractions defined in volume one and does not include specific language bindings. Volume three (ISO/IEC 10287-3) defines an interchange format (IIF) that allows CAI data to be stored or shared between applications. As of this writing, volume 2 (the PIKS API) has been formally accepted as an ISO standard. The other two volumes are still open for comment.

The PIKS API can be logically broken down into *data type* definitions, *tools*, *operators*, and *execution control* facilities.

Data types include images, arrays, lists, histograms, lookup tables, region-of-interest (ROI) descriptors. CAI images are logically five-dimensional arrays, with the indices conventionally interpreted as (x,y,z,t,b), i.e. volume, time, and spectral band.

Tools are routines that perform utility operations such as generating test images, filter kernels and so on.

Operators perform transformations between the various PIKS data types. They include a wide selection of standard image-to-image operators, plus statistics-gathering routines and a limited set of intermediate-level operators. Most operators allow images to be associated with a region of interest (ROI) and an index tuple called a match point. The ROI specifies the locations at which the operator is to be applied, while the match point specifies the logical origin of the image.

Execution control facilities are designed to allow PIKS operators to be executed efficiently on multiprocessor systems. They allow applications to request asynchronous execution of multiple PIKS operators, or to group sequences of PIKS operators into named collections which can be invoked in the same manner as primitive operators.

The PIKS operator set is comprehensive, including most of the important image operators in common usage today. We believe that it provides a useful guide to the sort of low-level computations AVIS users need to perform. The standards document (ISO/IEC standard 12087-2:1994(E)) provides formal mathematical definitions for each operator. Table 2 contains a summary of the op-

Table 2: PIKS Image Operators

operator name	description
global or region statistics:	
accumulate	sum pixel values over region
histogram 1D	compute histogram of pixel values
histogram 2D	compute co-occurrence matrix for specified offset
difference measures	sum of squared differences over region

Table 2: PIKS Image Operators

operator name	description
cross-correlation	normalized cross-correlation at a point
moments	mean and standard deviation of pixel values
extrema	max and min pixel values
pixelwise monadic operators:	
shift	logical, arithmetic or circular shifts on pixel values
LUT	map pixel values through a lookup table, with optional interpolation
monadic arithmetic	sum, product, difference, quotient, max or min of pixel values with a supplied constant pixel value
logical	bitwise AND, NAND, OR, NOR, XOR with supplied constant
scaling transformations	remap pixel values according to various laws: negate, square, cube, absolute value, threshold, clipping, gamma correct, erf, power law, rubber band, contrast stretch, histogram specification
color conversion	between standard color spaces or by user-supplied color transformation matrix or non-linear operator
display	false or pseudo color, ordered dither, Floyd-Steinberg dither
conversions	between integer, real, cartesian complex, and polar complex
pixelwise dyadic operators	
dyadic arithmetic	pixelwise sum, difference, product, quotient, max or min
dyadic logical	pixelwise/bitwise AND, NAND, OR, NOR, XOR
dyadic predicate	arithmetic relational operators
alpha blend	with constant or pixelwise variable alpha (e.g. chroma key)
pixelwise operators on multiple images:	
KL transform	pixelwise Karhunen-Loeve transform of multiband image
average	pixelwise average over any dimension: time, band etc.
Z-buffer merge	select one of two source pixels based on associated depth values
Neighborhood operators:	
convolution	2D or general 5D
template match	normalized correlation function

Table 2: PIKS Image Operators

operator name	description
median	median or rank filter
gradient	orthogonal or template-based with optional combination rule and thresholding
zero crossings	zero crossings of second derivative
morphological operators	boolean or grey-scale open, close, erode, dilate
hit-or-miss	replace pixels in a boolean image with any logical function of a 3x3 neighborhood
texture	Laws texture features
equalization	adaptive histogram equalization, Wallis statistical difference
pixel classification and labelling:	
Bayes or nearest neighbor	classify pixels in a multispectral image
slice	select pixels in some range of values
label	assign unique integer labels to connected components
Resampling and data movement:	
copy window	from subregion of one image to arbitrary position in another
zoom	expand with pixel replication
resize	expand or shrink with resampling
rotate	by an arbitrary angle about an arbitrary point, with resampling
simple rotate	flip, rotate by multiples of 90 degrees, transpose
polar	convert between cartesian and polar pixel indices
Region Operators:	
Note: most PIKS operators can take a region of interest as a control parameter, specified in any of several ways including via a bit mask. Thus most operators can be applied on a per-region basis.	
flood	flood-fill a region enclosed by an eight-connected boundary
perimeter	construct a crack code (analogous to chain code for four-connected images) for an eight-connected region
shape	given a region, computed standard shape metrics: perimeter, area, area inside perimeter, area of any holes, Euler number, circularity, bounding box, area ratio

Table 2: PIKS Image Operators

operator name	description
moments	given a region, compute Hu's invariant spatial moments
Global Operators:	
orthogonal transforms	Fourier, Hartley, Cosine, Hadamard
Hough transform	Using Duda and Hart angle/distance parameterization
linear filter	frequency-domain convolution. Note: kernel generators are provided for standard kernels, e.g. Butterworth, Gaussian etc.
homomorphic filter	frequency-domain convolution applied to log pixel values.

erator set. Tools, execution control facilities and housekeeping routines have been omitted for clarity. The PIKS execution control facilities are also of great interest. They are discussed in more detail in section 3.3, which deals with run-time system software.

3.1.4 VEIL

VEIL is a programming library for real-time image processing on heterogeneous pipelined image processors. It provides C++ class definitions that allow applications running on the host to describe image processing computations in terms of coarse-grained dataflow graphs. It translates these graphs to appropriate accelerator command sequences during program initialization, and allows the application to dispatch command sequences as coroutines.

The computational model that VEIL supports is a form of *homogeneous synchronous data flow* (HSDF) [30]. In this paradigm, each node of the dataflow graph produces exactly one token per arc per invocation. VEIL adds the restriction that all data sources are synchronous and fire simultaneously. Under this restriction the flow of tokens through the graph does not depend on the data, so a valid execution schedule can be computed statically.

The fundamental activity that VEIL supports is construction and execution of dataflow graphs. It provides the C++ programmer with two new object classes (data types): *graphs* and *operators*. Graphs correspond to single HSDF computations, each consisting of a set of operators (graph nodes) corresponding to image processing primitives. Each operator has some number of input and/or output *ports*, which are connected by *links* to form a graph. Operators also have attributes that specify details of their computation. Some attributes are common to all operators. For example, every input port on an operator has an attribute that specifies the size and relative position of the pixel array that the port processes. Most attributes are specific to individual operator types, however. These include such things as the masks for convolution or morphological operations, the contents of lookup tables, and the output scaling for arithmetic operations.

A VEIL application typically begins by creating a graph object and some number of operators. Next, the operators are inserted into the graph, their attributes are set and their inputs and outputs are connected to complete the graph. At this point the graph is ready for execution, and the program may proceed to create other graphs or perform any other initialization it requires. The program then invokes the VEIL scheduler, which verifies topological correctness and builds an

execution schedule for each graph. The scheduler arranges for the graphs to share hardware resources such as memory buffers, lookup table banks and kernel table entries whenever possible. Finally, the application executes the graph(s) by invoking VEIL's execution control methods.

The VEIL library supports rapid prototyping via an application called Nimue. Nimue is a dataflow graphical user interface that allows users to construct, run and modify VEIL graphs interactively. Graphs built with Nimue can be saved to files and later loaded into arbitrary C++ programs. The graph file remains accessible to Nimue. This allows programmers to modify the image processing that an application does without recompiling.

VEIL differs from most other dataflow rapid prototyping systems in its focus on developing efficient applications, as opposed to supporting interactive image processing and visualization. The Nimue graphical user interface is completely separate from the VEIL library, and has no more access to VEIL internals than any other user-level application. The advantage of this architecture is that it has encouraged the implementors to address the needs of general applications, rather than just those of the graphical user interface.

VEIL is also unusual in the way it integrates dataflow descriptions of computations into a conventional imperative language. From the point of view of the C++ programmer, VEIL graphs are completely ordinary objects, which can be stored in variables, copied, passed to procedures, or specialized by subclassing. In this respect they resemble the PIKS chain construct, which also encapsulates a (potentially) accelerated dataflow computation. PIKS chains and VEIL graphs differ in that the latter make the data dependencies explicit, and that VEIL's synchronization primitives operate at a higher level than PIKS' virtual registers.

VEIL and the sample tasks

VEIL differs from the other tools described here in that (at present at least) it is targeted at a specific hardware device: the Datacube MaxVideo system, which is a reconfigurable pipeline-based image processor. Thus VEIL can only perform those operations that the MaxVideo system supports. In general these are limited to simple image processing operations on eight- or sixteen-bit images. Its repertoire of operations is fairly limited, in part because the underlying hardware is extremely difficult to program.

For the Image Understanding Benchmark, the only operation that VEIL supports directly is connected component labelling of the intensity image. The depth image processing cannot be done in VEIL because the MV200 does not support floating point data. If byte data were used, VEIL could perform the required median filter, Sobel gradient and thresholding operations. All other operations would have to be done on the host.

For SLAM, VEIL is well suited to the image projection operations that dominate the recognition phase of the computation, since the MV200 can perform pixelwise multiplication and summing. New VEIL operators would have to be written to provide sixteen bit accumulation, but this presents no conceptual problem. VEIL would be of no help in finding eigenvectors or in the manifold computations, however.

VEIL's strengths are in real-time image processing, so it is not surprising that it is most useful for the hierarchical tracking task. In the actual implementation of the task, the Gaussian pyramids are formed using VEIL convolution and subsampling operations. The low-level coarse-to-fine search task is performed by the host, which loads the requisite image patches from VEIL output

graph nodes. VEIL is also used to perform all feature extraction and connected component labelling for the higher level tracking task.

3.1.5 VISTA

VISTA [39] is a software environment for computer vision research, created by Art Pope and David Lowe at the University of British Columbia. It is the latest in a series of University-developed vision toolkits that have been released via the Internet; previous examples include HIPS [27] and Improc [61]. VISTA is typical of the genre in that it is relatively small and limited in functionality, at least compared with commercial systems. On the other hand, its small size makes it easy to manage and gives it a short learning curve. VISTA is noteworthy for its clean design, excellent documentation, and focus on extensibility and higher-level data types.

The key design features and emphases of VISTA are:

an extensible file format. Many computer vision toolkits are restricted to low-level image processing on fixed image types. VISTA includes file types for edgels and line segments as well as the usual images. The file format is based on nested attribute/value pairs and is easily extensible, so that things such as camera matrices and history can be readily included.

a set of filter programs. VISTA provides a set of image processing programs encapsulated as UNIX filters, following the model of HIPS [27] and other image processing systems. This allows users to perform image computations interactively, or to construct programs quickly via shell scripts. Table 3 contains a list of the routines available in the current release of VISTA.

a C language API. Most of the VISTA filters are available as library routines callable from ANSI C. The library includes data type definitions, file I/O routines and access functions. The access functions have been carefully thought out and are quite efficient, reducing the temptation to write messy code that accesses the data structures directly.

support for X applications. The VISTA library includes a small set of routines for building interactive programs that run under XWindows. User programs can display images in windows, create dialog boxes, and attach callback functions to widgets or to mouse operations on images.

VISTA is quite small compared to some of the other systems described in this report, and its functionality is somewhat limited. In particular, the use of UNIX pipes for communication between programs is too restrictive for serious interactive programming. Systems that allow multiple inputs and outputs (e.g. Khoros) are significantly easier to use.

On the plus side, VISTA has a number of very attractive features. The file format and API are superior to those found in many older systems. Its focus on extensibility and higher-level data types are features that are badly needed. To date only KBVision [2] and the forthcoming IUE are superior to VISTA in this area. Finally, programs based on VISTA can be freely distributed and shared with other members of the vision community. This contrasts with systems such as Khoros, whose licensing restrictions make it difficult to distribute code that uses Khoros data types or facilities.

3.1.6 Ptolemy

Ptolemy is a software environment for simulating and prototyping heterogeneous systems, developed at the University of California at Berkeley. Like Khoros and VEIL (see above), it al-

Table 3: VISTA routines

routine	description
adjust	gamma correction and linear scaling of pixel values
convert	coerce pixel types with specified scaling/mapping behavior
crop	extract rectangular region
flip	flip image horizontally or vertically
invert	invert image intensities
istat	min, max, average and standard deviation
negate	negate pixel values
op	large collection of monadic and dyadic operators
rotate, scale	resampling with various interpolation strategies
transpose	image transpose
zeroc	zero crossings
convolve	space domain convolution
grad	gradient by central difference
gauss	separable convolution with gaussian or first derivative of gaussian
mag, phase	magnitude or phase of a complex image
complex	construct a complex image from real and imaginary components
fft	fast Fourier transform
(composition)	various routines for combining and separating image bands
(export/import)	various routines for convert to/from portable bit-map (PBM) format
(display)	X display and PostScript rendering
Canny	Canny edge detector (detection only)
link	Link Canny (or other) edges into chains with hysteresis
segedges	break edge chains into line segments by recursive splitting
calibrate	calibrate a camera by Tsai's method

lows users to build block diagrams whose blocks are computational entities and whose arcs are communications channels. It differs from them in having a much more flexible and general notion of block diagram semantics. As distributed, the environment is geared toward signal processing and communication systems, but it can also generate DSP code for hardware/software co-design.

The basic unit of computation in a Ptolemy simulation is the Block. Blocks play the role of procedures in conventional languages: they present a clean interface to a software abstraction which may be very complex internally. Typical blocks might represent an integrator in a signal processing simulation, or a service with associated queue in a queueing simulation. Blocks connect to the rest of the simulation through PortHoles, which are typed, buffered input/output ports.

In a running simulation, the execution of blocks is controlled by a scheduler. The scheduler is responsible for determining order of execution and invoking each block at the appropriate time. On invocation, each block typically consumes input, runs its internal computation to completion and produces output. Order of execution is generally based on the topology of block interconnections and on the state of the communications channels; however, the rules may be different in different schedulers. Schedulers may require particular behavior from the Blocks that they manage, such as a guarantee that the Block produces exactly one output message per outgoing PortHole. Thus Blocks and Schedulers cannot be freely interchanged.

A Scheduler plus the set of Blocks that it can manage constitutes a Ptolemy domain. Domains are an important and powerful concept, and are the primary difference between Ptolemy and other data flow programming systems. Within a domain, communications semantics and scheduler behavior are uniform and obey a single computational model. Domains supplied with Ptolemy include synchronous data flow, dynamic data flow, discrete event and queueing. Multiple domains can coexist inside a single simulation, as described below. This allows each subsystem in a complex simulation to be built using the most appropriate simulation model for the problem.

Structure of a Ptolemy Simulation

Ptolemy Blocks are specialized into types whose names follow a cosmological metaphor. Atomic or primitive Blocks are called Stars, and are implemented in native code (C++ for the supplied domains) for the target machine. A Galaxy is a block consisting of a set of interconnected blocks. It can be thought of as a macro that expands to a network at simulation time. Blocks within a Galaxy must all share a common domain, which is inherited by the Galaxy as a whole. Finally, a block may be a Universe, which consists of a set of blocks plus a Scheduler implementing some domain.

Multiple domains may be included in a simulation by means of the Wormhole construct. A Wormhole is a subclass of Star and is managed as such by the Universe it occupies; thus it must obey the communications semantics of the associated scheduler. Internally, however, the Wormhole may contain an entire Universe implementing a different domain and hence a different computational model. The Wormhole code is responsible for making sure that each domain sees consistent communications behavior at its inputs and outputs.

Ptolemy schedulers are aware of the architecture on which they run. Scheduling itself is abstracted into an initialization phase and a run-time phase. This provides a tremendous opportunity for exploiting special hardware. For example, in restrictive domains such as synchronous data flow a legal order of Block execution can be determined statically. A scheduler for such a domain can determine the order during initialization, perhaps distributing tasks over multiple processors and generating a static executable to perform the computation. The run time component of the scheduler simply invokes the compiled code. Scheduling for synchronous data flow is explored in [29], and is also implemented in VEIL (see above).

Ptolemy is fundamentally a C++ library and is less dependent on a graphical user interface than are systems like AVS [56]. It does provide a Tcl/Tk based interface called Pigi, which communicates with the design database Oct through the Ptolemy kernel. Pigi follows the manipulation conventions of an earlier generation of CAD tools, and is frustrating to use for programmers trained on direct manipulation systems such as Khoros, AVS, VEIL and so on.

Ptolemy and the Sample Tasks

Ptolemy is not an image processing system per se, and does not support the sample tasks to the extent that the IUE or Khoros do. Ptolemy does offer interesting possibilities as a way of structuring large computations such as the Image Understanding Benchmark. The messages that pass between Ptolemy blocks are instances of the abstract class Particle. Thus although Ptolemy does not provide any vision-related data structures beyond basic images, there is no conceptual barrier to building blocks that pass around (for example) IUE Spatial Objects. This would allow it to handle descriptions of the form shown in figure 2. In the IUB task, early image processing could be handled by the synchronous data flow domain, while processing of chain codes and regions could be handled by dynamic data flow. Dynamic data flow could also handle the iteration over sets of images that is required for SLAM. The real-time character of the hierarchical tracker would be well suited to a synchronous data flow domain. The current implementation of the tracker is in fact implemented using VEIL, which supports a special case of synchronous data flow.

3.2 Low-level Development Tools

The previous section reviewed tools that allow users to construct applications by combining existing computational primitives (e.g. subroutines or programs). In this section we turn to the tools needed to develop primitives, and more generally to build the systems support necessary for higher-level development environments. In general these tools will consist of compilers, debuggers and profilers, but will likely include features and/or runtime support not usually found in compilers for conventional architectures. In the discussion that follows we assume that conventional compiler technology is available, and focus on the extensions or support libraries needed to support accelerators of various types.

We believe that the higher-level tools discussed in the previous section can and should conceal as much as possible of the underlying hardware. By definition, a library routine is meant to be called in many different contexts on many different arguments. If it is tailored to a particular machine configuration or calling environment, it loses its generality and much of its usefulness.

In contrast to development environments and libraries, we expect that tools at the lower level will probably have to expose the hardware to a considerable extent. This will be most true for highly specialized architectures in which much of the work is done by dedicated functional units. More conventional and general devices (e.g. MIMD machines) will be able to provide multiple levels of software abstraction, so that programmers building library routines can code in terms of an idealized parallel machine.

Because we expect low-level tools will work intimately with the hardware, solutions at this level will be different for different architectures. In the remainder of this section we discuss available technology and approaches for MIMD architectures, SIMD machines and configurable pipelines.

3.2.1 MIMD and SPMD architectures

MIMD architectures consist of computing nodes (typically a conventional CPU with some amount of local memory) connected via some sort of high-speed communications network. The network may provide the illusion of shared memory, but more often supports a messaging system that transmits packets between processes. The latter type of MIMD architecture has come to dominate the market for large-scale multiprocessors and is widely considered to offer the only practical way to build very high-performance, general purpose, scalable machines.

There are currently three dominant approaches to providing low-level tools for MIMD message-passing architectures. The simplest is to provide access to the hardware messaging system and process creation via operating system and library calls. This exposes the hardware more than one would like, but is low cost and provides access to the full power of the machine. An alternative is to provide some level of language support that allows the programmer to express 'natural' partitions of the task in a graceful manner, and let these tasks be distributed automatically by the language's runtime system. Here we will refer to systems that take this approach as *distributed object* systems. Finally, the third option is to emulate an SIMD architecture and use the programming tools (e.g. data-parallel languages) designed for those machines. Automatic compilation of arbitrary codes to MIMD machines is an open problem with no practical solution in sight.

Message Passing Systems

Message passing systems [33] are currently the most widely used method of programming large-scale parallel processors. They have been particularly successful in scientific computing applications. Most MIMD vendors provide proprietary packages for their machines, and there are several portable systems available as well. Because the functionality they provide is conceptually very simple, they require no language or compiler support, and typically take the form of C- or Fortran-callable library routines.

The range of semantics that a message passing system can have is limited, so all message passing systems present similar-looking interfaces. The most widely distributed message passing system is the Parallel Virtual Machine (PVM) [51], which is typical of the genre. PVM was designed to support applications distributed across heterogeneous local area networks, but is extremely portable and has been used on more tightly coupled parallel machines as well. The PVM library allows programs to create tasks and associate them into groups. Processes communicate by single or multicast non-blocking Send and blocking or non-blocking Receive. Receive calls can filter the incoming message by requesting only messages from specific source tasks and/or messages bearing specific user-defined tags.

One problem with PVM-like systems is that they make it difficult to write safe library routines. Programmers are permitted to use 'wildcard' descriptions for the source and tag fields of Receive calls, so it is easy for a library routine to accidentally receive a message intended for its caller. Avoiding wildcards helps, but tag fields are user-defined and form a global name space, so tag name clashes are always a possibility.

In order to improve the portability of applications between multiprocessors, a consortium of vendors and academic researchers have agreed to develop a standard message passing interface called MPI [57]. MPI is designed to exploit the features of a variety of MIMD architectures, and includes extensions (derived from the experimental ZipCode system [49]) intended to support creation of safe parallel libraries. MPI's key features include:

- Messages may be sent in any of several modes, which provide various restrictions which the receiving process must meet for the send to succeed. This allows messages to provide various degrees of light-weight synchronization.
- Processes are static; it is not possible to create a process during program execution. However, processes may associate in groups which can be altered dynamically.
- Library routines can declare a *context*, which is a dynamically created and guaranteed unique tag associated with each message sent by a context member. Messages cannot pass from one process to another unless both parties are in the same context, so name conflicts and unintended message interceptions can be prevented.
- Groups of processes are organized into topologies such as grids or arbitrary graphs, and can address their neighbors using the appropriate topological identifier. Thus it is never necessary for a process to refer to a physical processor or process ID number.
- A small number of specialized services are provided: large data objects may be distributed across process groups using broadcast and scatter-gather messages, and transformed using parallel *reduce* and *scan* operations.

The MPI specification has not been finalized, and it is likely that details will change. It is stabilizing rapidly, however, and is likely to become a standard well before the end of the AVIS program.

One feature of message-passing systems that has not made its way into MPI is the idea of *active messages* [14]. Active messages are designed for use in a single-program, multiple-data (SPMD) context. In an SPMD program each processing node has a copy of the same executable, so global objects (notably procedures) can be referred to by their addresses. An active message consists of a pointer to a procedure followed by the arguments that are to be passed to it. They can be used as a substrate for implementing higher-level communications protocols, or can be used directly by applications programmers as a way to distribute tasks. They were used very effectively on the CM-5 [54] to provide extremely low-overhead messaging between user processes.

Shared Objects

Message-passing systems have proven to be an effective way to program MIMD machines, particularly for large scientific applications. Their main drawback is that they provide very low-level access to parallelism, requiring the programmer to handle data distribution and synchronization explicitly. The result is that building parallel applications is painful and error-prone. As an alternative, a number of programming systems provide parallelism via operations on distributed data structures. Synchronization is hidden (to varying degrees) in accesses to these data structures, or in operations on them.

The Linda programming system [10] is one of the best-known distributed object programming systems. Linda provides processes with access to a shared associative memory called tuple space. Library calls allow tuples containing data to be stored or retrieved based on partial specification (i.e. by providing values for some of the tuple entries). In many cases parallel programs can be written without any need for explicit process ids - the associative tags associated with data objects provide all of the identification necessary. Linda can support a number of programming paradigms (such as agenda-based parallelism) that are hard to express in message-passing systems. Commercial C and Fortran-based implementations of Linda are available.

The principal difficulty in implementing Linda lies in providing efficient implementations of the tuple space operations. Carriero and Gelernter [10] describe methods of detecting many tuple matches at compile time, so that expensive run-time associative searches can be avoided. Repetitive communication patterns can also be detected at run-time and used to dynamically optimize the processing of tuple space transactions, so that the number of interprocessor messages is no worse than would be expected in a conventional message-passing implementation.

Mentat [15] is an object-oriented programming language for MIMD architectures. It is an extension of C++ and differs only slightly in syntax from the base language. It does not support any notion of explicit processes. Instead, programmers can declare mentat classes whose methods are implicitly called in parallel. The Mentat compiler analyzes the graph of data dependencies involving mentat class instances and inserts appropriate synchronization. For example, in the simplest case the invocation of a method on a mentat object is transformed into a non-blocking remote procedure call to a compute server on some remote CPU. The compiler detects the first use of the result in the caller and inserts a blocking receive at that point. If the result of the invocation is passed as an argument to another implicitly parallel operation, the runtime system will arrange for the result to be forwarded to the CPU responsible for the second operation, so that the data does not have to be transmitted twice. In effect, the run-time system dynamically constructs a dataflow graph representing the computation, and does not block the calling process until the result of the graph computation is used locally.

3.2.2 SIMD architectures and Data Parallelism

In many programming problems the obvious unit of parallelism is the individual data element in a larger structure. For example, in image convolution each output pixel is a weighted sum of pixels in the corresponding neighborhood in the input image. Convolution can be parallelized by assigning a virtual processor to each output pixel, and instructing the processors to fetch their neighborhoods, compute their results and store them, all in parallel.

Data parallel languages are designed to make it easy to express computations involving parallel operations on each element of a data structure. The term 'data parallelism' was introduced in the context of programming a SIMD architecture [19], but the underlying ideas have a long history [48]. Data parallel constructs provide a very natural way to express low-level vision algorithms, but they have also been shown to be surprisingly well suited to more general computations [20]. Programs based on data parallelism are often much easier to understand and maintain than programs based on communicating processes. For this reason there has been considerable interest in supporting data parallel languages on non-SIMD architectures.

The basic capability provided by data parallel languages is the ability to perform operations in parallel on every element of a data structure. For example, a language might allow the programmer to multiply an array of numbers by a scalar, and might implement the operation by performing all of the multiplications in parallel. Sipelstein and Blelloch [48] review issues in the design of data parallel languages, which include:

what structures can be operated on in parallel? Arrays and vectors are supported by most languages; lists, sets, and other structures (e.g. paralations) may also be useful. A major question is whether the structures may contain arbitrary data types, be heterogeneous, and/or include nested parallel structures. Allowing for these possibilities makes the language more general but raises subtle issues of interpretation.

what operations can be performed on the structures? Operations may be limited to arithmetic primitives, or may allow calling arbitrary user functions. If operations can be arbitrary and nested parallel structures are allowed, the language effectively becomes control-parallel.

what domain is supported? Although many data parallel languages are intended to be completely general, others designed for specific classes of problems. Some are explicitly designed for particular domains, including image processing.

Array-oriented languages

Scientific computation is a major target of parallel language designers, so many data parallel languages are optimized for parallel array operations. Fortran 90 [48] is a typical example. It allows the programmer to declare parallel arrays of varying dimension. Arrays may contain only numbers; nesting is not allowed. The language allows primitive monadic and dyadic operations on arrays and between arrays and scalars, as well as reduction operations such as Max, Min, Sum et cetera. An array can be used to index another array, resulting in a parallel permutation.

The array is also the basic unit of parallelism in C* [43] and its descendant, Dataparallel C [18]. These languages are significantly less restrictive than Fortran 90. They permit arrays of user-defined types, and allow users to define functions that are applied to array elements in parallel. They do not support nested parallelism, however, so code that is applied in parallel cannot itself invoke parallel operations.

General languages

A number of experimental languages have been developed to explore the limits of parallel data structures. The language SETL [47] allows parallel operations on ordered and unordered sets, where set contents are arbitrary, nested sets are allowed and parallel operations can invoke other parallel operations. Paralation LISP [46] defines a data type called a paralation, which can be thought of as a relational database. Both languages allow operations to be applied across all elements of a set or paralation, or to be applied only to those elements satisfying some test.

Higher level data parallel languages such as paralation LISP raise interesting issues in language design and compiler technology. At present however they are fairly experimental and it is not clear whether AVIS applications programmers would find them usable.

Languages for Image Processing: APPLY and ADAPT

APPLY [17] and ADAPT [59] are a pair of languages designed specifically to support low-level and intermediate-level computer vision algorithms. They were originally developed as tools for programming the Warp [3] series of systolic computers. However, the computational model they support is quite general, and compilers have been designed for a variety of architectures including shared-memory machines and hypercubes.

APPLY and ADAPT are each designed to support a single, highly specific programming style where data dependencies follow a particular pattern. This allows them to be translated in a straightforward way into a stereotyped pattern of data distribution and computation steps. The compiler's task is to analyze the program description and determine values for any free parameters in the pattern.

APPLY supports operations in which every output pixel in an image is a function of a corresponding neighborhood in one or more input images. The programmer expresses the computation by writing a procedure in an ADA-like language. The procedure header declares the input neigh-

borhood size, output data type and any auxiliary parameters or local storage required. The procedure body consists of a block of code that describes the computation for a single output pixel, assuming an output coordinate of (0,0). The neighborhood-to-output address mapping can be modified so that the neighborhood operator is applied only at points of an integer sampling grid, or so that multiple output pixels are generated for each neighborhood. This allows the up- and down-sampling operations needed for pyramid construction.

ADAPT supports operations in which the output is a data structure which in some way captures properties of the entire image. It allows a degree of data dependence, in that neighborhood operations can make use of values computed during the previous neighborhood along the same row. It provides for user-supplied code that combines partial results of image subregions to form a result for the union of the regions. The paradigmatic example of such a computation is the image histogram: a histogram can be computed by partitioning the image, computing the histogram of each partition, and then returning the elementwise sum of the partial results.

ADAPT syntax is similar to that of APPLY, with additional keywords and parameters to support its somewhat different semantics. ADAPT procedures consist of an APPLY-like header followed by four blocks of code labelled FIRST, NEXT, COMBINE, and LAST. The FIRST code is executed before any pixels have been processed. It is used to initialize any storage, e.g. to zero out an array of histogram accumulators. The NEXT code is executed once for each pixel in the image, implicitly in raster-scan order. At each invocation, the NEXT code can refer to any local variables, which will have the values they were assigned during the FIRST code or the previous invocation. The COMBINE code is called to merge results computed by the NEXT code for two subregions of the image. It can refer to the values of lOcal variables for either subregion. Finally, the LAST code is called to take the combined results for the entire image and store them in the procedure output parameters.

APPLY and ADAPT are not full-fledged programming languages. They do not allow programmers to define anything corresponding to the main program of a conventional application. Their function is restricted to defining procedures which follow one of the two computational models defined above. These procedures are meant to be linked with and called by an application written in a conventional language (e.g. Common LISP). This is a natural programming model for accelerators which depend on a host computer for sequencing and control, such as the WARP.

Although the computational models underlying APPLY and adapt are quite restrictive, they can handle almost all of the standard low-level image operators defined in the PIKS standard, as well as many of the other computations discussed in sections 2 and 3.

3.2.3 Configurable pipeline architectures

There are a number of image processing accelerators on the market that do not fit the conventional SISD/SIMD/MIMD characterization. These systems are typically programmed by having the host computer dynamically configure the hardware to form a circuit that performs the desired computation. We refer to these systems generically as configurable pipelines; examples include the ASPEX Pipe [23] and the Datacube MaxVideo series [13].

Configurable pipeline architectures are programmed by having the host computer store appropriate values into registers. For DataCube products, the ImageFlow library [12] automates some of the register computations, for example by automatically computing pipeline delays. The

state of the art in low-level tools, however, is represented by graphical microcode assemblers. These are programs that present the user with a graphical representation of the data paths and computational elements of the machine. The user specifies the flow of data through the system by clicking on datapath elements with the mouse. The first interface of this type was ASPIPE, a graphical assembler for the Pipe. Datacube now markets a tool called WitFlow that appears to offer similar functionality.

Graphical assemblers are a tremendous improvement over text-based interfaces to architectures of this type, because they free the programmer from having to look up the names of the datapath elements in manuals. They are still low level tools, however, because they require the programmer to describe the computation in terms of what the components of the machine should be doing at each particular moment, rather than describing what should happen to the data.

3.3 Runtime environments

Runtime environment software provides the interface between the raw hardware and user code generated by low-level tools. The hardware and runtime environment software together constitute a virtual machine that automates such low-level functions as downloading code to the accelerator, controlling its execution, and managing communication between accelerator and host.

Because it is so close to the hardware, runtime environment software depends critically on details of the accelerator it is designed for. In general it may consist of one body of code that runs on the host and another that runs on the accelerator, and may be divided into layers providing different levels of service.

All types of architectures will require some sort of host-side device driver to control access to the accelerator. The driver must at a minimum allow host programs to transfer data to or from accelerator memory, download accelerator executables, and run, halt or synchronize with processes running on the accelerator. Accelerators that have substantial general-purpose computing power on board (e.g. MIMD machines or machines with an embedded processor for control and sequencing) will need additional runtime software for the on-board processor(s). In simple cases this may consist of no more than is necessary to accept executables from the host and run them. More complex systems may require a full-blown operating system.

The technological problems posed by AVIS accelerators are relatively modest at this level. Addressing them should be well within the state of the art. On the other hand, few if any existing products will meet the needs of the AVIS program without modification. In this subsection we review systems that appear to be good models for AVIS runtime systems.

3.3.1 DSP Operating Systems and APIs

A number of vendors have developed operating environments for DSP microprocessors. The goal of these systems is generally to allow programmers coding for conventional workstations or PCs to make use of attached DSP-based accelerators in an easy, natural and portable manner. Although these systems are unlikely to be useful to the AVIS program in their current form, they provide a useful guide to the sort of capabilities that AVIS applications programmers might need.

Visible Caching Operating System (VCOS)

VCOS™ is an operating system and DSP toolkit written by AT&T for use with the DSP3210 microprocessor. It consists of a set of OS routines that run on the DSP chip, a library of modules that perform standard multimedia functions (such as implementing a V.32 modem), and a well-defined API that permits the host programmer to create and control DSP computations.

VCOS permits host applications to instantiate one or more tasks consisting of multiple modules communicating through buffers. It allocates resources (including CPU time) among the tasks and declines to create tasks whose resource requirements can't be met. Each module performs a single transformation on a data stream, such as compressing a stream of speech. Modules may declare themselves to be real-time, in which case they will be guaranteed a fixed portion of each 10 millisecond time slice. The remainder of each time slice will be devoted to non-real-time tasks. Buffers may be any of several functional types with differing semantics, such as FIFO buffers for time series data, or parameter buffers for module control information. The buffer mechanism is used both for communication between modules and for communication with the outside world. For example, a FIFO buffer may be used to send audio data to an amplifier and speaker for playback, or to allow an application running on the host to send image data to a DSP-based image filtering module.

SPOX

SPOX™ is a proprietary DSP development and delivery environment distributed by Spectron Microsystems, Inc. Unlike VCOS, it is intended to be portable and runs on most of the currently popular DSP microprocessors. SPOX consists of three separate components. SPOX-DSP consists of development tools (compilers and linkers) and an optimized library of matrix, vector and filtering operations. Its purpose is to allow programmers to create DSP routines that run on the target processor. SPOX-RTK is a real-time operating system kernel that provides the execution environment for SPOX-DSP programs. It includes facilities for multitasking on the DSP processor, and for interrupt-driven communications with external devices such as sensors and displays. Finally, SPOX-LINK allows DSP devices that are attached to PCs or workstations to communicate with their hosts and perform system operations such as file I/O. It also allows RPC-like invocation of DSP operations by applications running on the host.

3.3.2 PIKS Execution Control Facilities

The Programmers Imaging Kernel System (PIKS) [21] is an API for computations on images and image-related data structures. The basic architecture of the system and the image operator set were presented in section 3.1.3.

PIKS is specifically designed to be supported efficiently on symmetric multiprocessors, distributed systems, and computers with attached accelerators. It does so by defining a model of execution and execution control that makes coarse-grained parallelism readily visible, and allows the application to request parallel asynchronous execution.

PIKS does not suggest implementation mechanisms, so it is not 'available technology' in the strict sense. It is however a very well-designed and eminently supportable standard. We believe that it provides an excellent model for AVIS run-time software.

The PIKS programming model

All PIKS programs manipulate PIKS data objects, which are created by calls to the PIKS API. The simplest possible sort of PIKS program would be one which creates a few image objects, loads data into them, invokes a series of image processing operators on them, and finally reads back the results. By default, all of these operations are performed in synchronous mode. In this mode each operator has the semantics of a procedure call; execution of the operator is guaranteed to be complete when control returns to the calling process.

A simple form of parallelism can be obtained by setting the global execution model to asynchronous mode. In this mode, operator invocations are allowed to return before the requested operation has completed. Instead of blocking, each API call that requests an operation has the effect of enqueuing the request inside the PIKS runtime system. The runtime system will then execute the requests in FIFO order. The application may perform other computations in (logical) parallel with the PIKS computation. Operations which return a result to the caller will block the application until the queue is empty, enforcing synchronization. An explicit Synchronize() operation is also available.

More sophisticated synchronization capability is provided by means of virtual registers (vregs). Vregs are typed dynamic variables capable of holding one PIKS data object. They can be used just like other PIKS data objects, but in addition have the semantics of semaphores. An attempt to read a vreg that is in a CLEARED state has the effect of blocking the reader until the vreg has been set. This is true regardless of whether the reader is the application or an operator. For example, suppose that the application requests asynchronous execution of some operator P that takes an integer parameter. If the request supplies an integer vreg as the parameter value, the runtime system will execute everything preceding P in the queue and then block waiting for the vreg. When the application supplies a value for the vreg, operator P will be released and execution will continue. Similarly, the application can block on vregs that serve as operator outputs.

The most powerful opportunity for exploiting parallelism in PIKS is provided by the *chain* mechanism. To create a chain, the application calls ChainOpen() and then makes operator calls as usual. Instead of executing or enqueuing the operators, however, the runtime system simply adds them to the chain. When the chain is closed, it becomes executable; calls to ChainExecute() initiate asynchronous execution of the operators in the chain.

The PIK standard specifies that the result of executing a chain shall be the same as if the operators were executed in the order in which they were entered into the chain. However, since PIK operators are side-effect free, all data dependencies between operators are implicit in their input and output parameter lists. This allows the PIK system to find the data dependencies and construct a dataflow graph corresponding to the operator sequence. If hardware support is available, it can then exploit any opportunities for operator-level parallelism that the dataflow graph contains. Another benefit of the chain mechanism is that it allows implementations to optimize sequences of operations. For example, an implementation that represents large images in a tiled format might choose to pipeline the operators so that the each tile is read from disk only once, rather than being written out and read back in again between each pair of operators.

The vreg mechanism interacts with the chaining facility to provide additional opportunities for synchronization and parallelism. A trivial example would be the use of vregs to pass an intermediate result from a running chain to the application, or to allow the application to supply new parameters to a running chain. The chain control functions allow a chain to be executed inside an-

other chain, optionally conditioned on the value of a vreg. An operator is also provided that can increment the value of a scalar-type vreg. Together, these mechanisms allow chains to contain the equivalent of FOR loops, WHILE loops, and conditional and CASE statements.

The PIKS execution control facility does have shortcomings. Most important of these is the fact that there is no explicit provision for interacting with external devices such as cameras. Real-time vision applications will need some such mechanism in order to allow computation to be triggered by the arrival of a new video frame. The simplest method would be to provide an additional operator that reads from a hardware digitizer and stores the result in a vreg.

Another problem with the PIKS model concerns the treatment of time. In many applications it is useful to think of the operators as long-lived objects with internal state, which perform computations on a potentially infinite series of inputs. For example, a temporal FIR filter might be conceptualized as a procedure that maintains a static ring buffer containing the k most recent images. On each invocation it accepts a new input image and produces a pixelwise weighted sum of the images in the ring buffer. PIKS operators do not have internal state and hence cannot be used in this manner. It appears possible to implement the same functionality using virtual registers: in particular, the CopyWindow() operator could be used to store successive inputs into successive subarrays of an image of temporal dimensionality k. Both the image and the temporal index would be implemented with virtual registers. However, the mechanism is clumsy and it is not clear whether the standard is explicitly intended to support this 'systolic' style of computation.

3.3.3 The ImageFlow Device Driver

ImageFlow is the collective name given to a set of software development tools for image processing boards manufactured by DataCube Inc. of Peabody, Massachusetts. ImageFlow consists of a set of libraries that can instruct the hardware to perform computations, accompanied by a device driver that controls the way in which those computations are executed. The device driver is extremely sophisticated and demonstrates an interesting and potentially useful way of controlling high-performance image processors.

In order to understand the device driver it is necessary to know a bit about the computational capabilities of the hardware it supports. At the hardware level, DataCube systems can be thought of as consisting of a heterogeneous collection of processing elements and memory buffers connected by an irregular network of dedicated communications channels. The memory buffers are dual-ported and contain programmable address generators, which allow them to read or write two-dimensional arrays of pixels in raster-scan order with programmable array size and stride. The processing elements include ALUs, hardware convolvers, look-up tables and so on, and are designed to work on data presented in raster-scan order. The communications channels include multiplexers and crosspoint switches, allowing data to be routed to different functional units under program control.

DataCube systems do not contain conventional CPUs. To perform a computation, the host computer (a Sun or SGI workstation) loads a series of control registers that define the access pattern for the address generators, the connectivity of the communications channels, and the operations to be performed by the processing elements. It then 'fires' the computation, causing the address generators to begin transferring pixels to or from the processing network. If the hardware configuration includes a video digitizer, the computation can be synchronized to the video input

and can be repeated indefinitely without further intervention by the host. Typically, however, the host waits for the end of the computation and then steps in to reconfigure and restart the device.

The ImageFlow device driver plays several roles in the process described above. These include the traditional ones of controlling access to the device and providing safe transfers between the virtual address space of the application and the physical one of the device. Its most interesting function, however, is to allow sequences of ImageFlow computations to execute without intervention by the host application, with synchronization when the application requires it. In essence, the driver allows the application to upload descriptions of multiple computations, and to create a finite state machine that controls their execution. State changes can be triggered by the completion of a single computational step, by the results of computations on the hardware, or by events posted by the host application. The state machine can also generate events that the application can receive, allowing synchronization when it is needed.

The net effect of all of this mechanism is to provide functionality similar to that of the PIKS execution control facility (see above). Host applications can create multiple computational processes that communicate with each other and the host through buffers. The host can trigger a series of computations by storing data into a buffer itself, or by attaching a buffer to an external data source such as a video digitizer. The host can alter the parameters of a running computation, and can synchronize with the computation as needed using a semaphore-like event mechanism.

4 Recommendations

The goal of the AVIS program is to produce hardware accelerators that provide an order-of-magnitude improvement in productivity for engineers and scientists working on problems involving imaging. This report is intended to provide guidance on the question of what sort of software should be provided with AVIS systems, with particular attention to the needs of researchers in image understanding, automatic target recognition and real-time computer vision. Previous sections of the report surveyed the computational and practical needs of the intended users, and examined the software technologies available for addressing those needs. In this section we draw conclusions from these surveys and make specific recommendations about the types of software that AVIS system designers should supply.

The recommendations presented here were arrived at by balancing a number of concerns. The primary driver was our analysis of what the intended users need in order to be productive. These needs are defined by the types of computations users perform, by their working environment, and by the goals of their research projects. Another consideration was the state of the art in software tools for conventional computers. Users are unlikely to adopt AVIS systems as development tools if the environment they provide is dramatically worse than what is available on their workstations. Finally, our recommendations were influenced by practical considerations of cost and technological risk. Providing a software suite that is at good enough to be useful will require a substantial development effort, and will stretch current software technology to its limits. We have done our best to identify a solution that is capable enough to be useful, but simple enough to be implementable at reasonable cost and without going too far beyond the state of the art in software technology.

Although this section draws heavily on material presented in previous sections of the report, it is self-contained and does not assume familiarity with the rest of the document. It begins with an executive summary of the domain needs survey of section two. We then restate user needs in terms of a set of abstract goals to be met and functionality to be provided. This is followed by specific recommendations about how to achieve those goals. Finally, we discuss areas of technological risk and their implications for the development process.

4.1 Domain requirements

AVIS accelerators are expected to be useful in a wide variety of application areas. For the sake of concreteness, however, this study is limited to the domains of image understanding, real-time computer vision and automatic target recognition. Section two of this report examined the needs of researchers in these areas in detail. Here we briefly summarize the results of that inquiry.

The domains in question are all areas of active research. Many unsolved problems remain, but in all three domains applications are beginning to move out of the laboratory and into fielded systems. In certain niche applications they are likely to have substantial commercial and military impact within the next few years. At the moment, however, all three domains are highly experimental, and the primary activity of users in the domains is research and development. It is this activity that AVIS accelerators must be designed to support; they should *not* be conceived of primarily as application delivery vehicles. Software development tools must be an integral part of AVIS systems.

To first order, all three domains perform the same types of computations and rely on the same basic computational paradigm. Their computations can be divided into three basic categories. *Low level operations* are those that map images to other images, or to statistical descriptions of images. They include noise removal, filtering, and certain types of feature extraction and reconstruction operations. *Intermediate level operations* construct a description of the image in terms of extended structures such as lines, regions, surfaces, or geometric primitives. Finally, *high level operations* examine the relations between groups of intermediate level descriptions for purposes of recognition and/or geometric reasoning. Examples of important computations of each type are given in sections two and three of the report.

Within the general paradigm presented above, there are significant differences of emphasis. Automatic target recognition systems must often make decisions based on low-resolution images with relatively few pixels on target. This has historically led them to stress low-level computations and use recognition algorithms based on statistical pattern classification. The recent trend, however, is toward model-based approaches, which use a broader range of computer vision techniques. Researchers in real-time computer vision appear to be following a similar path for different reasons. Older real-time vision systems emphasized low-level computations because of their generally deterministic running time, and because the need for real-time response left no time for anything else. Progress in the design of agent architectures (together with faster computers) has begun to permit much more high-level processing, and the question of how to integrate processing at different levels is now of great interest.

Where the domains of interest differ most is in the environment in which the computations are performed, and in the data rate and size of the images they must process. Some image understanding systems (e.g. for medical or aerial image analysis) must handle enormous images with tens or hundreds of millions of pixels. Generally in such systems the images cannot be memory-resident, so processing must be performed on a block basis. Running times and latencies are determined by database and I/O performance and are typically long. At the other extreme, real-time computer vision and tactical ATR systems process relatively small images (e.g. 512x512 for a typical video camera), but must handle thirty to sixty frames per second with latencies on the order of a frame time.

The primary working environment for most researchers in the domains of interest is the networked UNIX workstation, with PCs beginning to play a role on the commercial side and as delivery environments. The dominant programming language is C/C++, with a significant LISP/CLOS community. On top of the low-level languages most researchers use some type of higher-level development and execution environment. The best known of these is Khoros [24], which has become standard and is now required for many ATR contracts.

4.2 Software Goals

In the course of the domain survey summarized above we talked to a large number of potential AVIS users and looked at many applications. These experiences led us to a number of broad general conclusions about the type of functionality that AVIS software must provide. The most basic of these is simply that *the application developer is the most important customer for AVIS systems*, and that serving the developer's needs is the most important function of AVIS software. This principle follows from the fact that research and development is still the primary activity of

potential AVIS customers. It is of course to be hoped that AVIS systems will enable interesting applications and eventually be sold to application users as well as developers. We believe, however, that there is absolutely no chance that this will happen unless the development tools are of very high quality.

The goal of making developers happy is too broad to serve as a useful guide in designing development environments for AVIS systems. We have identified four critical characteristics that we believe AVIS development tools must have in order to achieve the higher goal:

Supporting Code Reuse

Code reuse is a topic of great interest in software engineering circles [26]. It is generally accepted that designing for reuse can lead to both improved programmer productivity and increased reliability of the resulting code. If code reuse is important for software development on conventional architectures, it will be doubly so for AVIS systems. It is a trivial observation that writing code for AVIS systems will be at least as hard as coding for conventional computers. For many architectures it will certainly be harder. Thus, the fewer lines of new code AVIS developers have to write, the happier they will be.

Supporting Rapid Prototyping

We claim that AVIS application developers are for the most part engaged in research and development. This means that they spend a relatively large fraction of their time building new applications and studying their behavior, compared to the time they spend actually running the applications they have built. This being the case, it is important to provide rapid prototyping tools that make it easy to put together a new application and experiment with it under different conditions. Systems of this type already exist for conventional workstations (e.g. Khoros, see section 3). Applications developers will expect tools of comparable quality for AVIS systems, and will resist using the systems if the tools are not there.

Supporting System Integration

Most standard ways of supporting code reuse and rapid prototyping place restrictions on the forms that code modules and programs can take. For example, visual languages like Cantata (the Khoros GUI) generally take over the flow of control from the application, and specify the form an image processing operator must have in order to be usable via the visual programming interface. The danger is that these restrictions may make it difficult to embed code developed with the rapid prototyping system into a larger application. In the case of Khoros, for example, it is quite difficult to embed a Cantata computation in a larger program written in C or LISP.

AVIS development tools must provide the programmer with more flexibility than is available in most workstation-based rapid prototyping systems. Real-time computer vision applications in particular will need to be able to interleave computations running on the AVIS accelerator with control and communications activities performed by the host computer. The rapid prototyping tools must not be so rigid as to preclude this.

Making efficient use of the hardware

Speed of computation is of extreme importance to many AVIS users. All of the domains of interest require high throughput, and some (notably real-time vision and ATR) also have severe latency requirements. In these cases speed may be as important as more traditional performance measures (precision, receiver operating characteristic etc.) in evaluating the suitability of a given

algorithm. It is critical that AVIS development and rapid prototyping tools provide users with a realistic idea of how quickly their algorithms are likely to run in the context of an application. The development tools should also provide a smooth path for incremental performance enhancement. Users should never have to recode an entire prototype application from scratch in order to speed it up. Instead, they should be encouraged to profile the application and recode just enough of it to achieve the required level of performance.

One disadvantage of many software development environments is that they achieve ease of use at the cost of substantial overhead for task creation and dispatching. For workstation-based systems the cost is bearable because it is dwarfed by the cost of performing the image computations. For AVIS systems this will no longer be the case. AVIS development tools must be carefully designed to ensure that the speed gained on the image computations is not lost again at the control and sequencing level.

4.3 Achieving the Goals

Achieving the general goals of code reuse, rapid prototyping, support for system integration, and efficiency will require a substantial amount of work. Exactly how the goals should be achieved depends on several factors. The first consideration is the accelerator and system architecture. Designers will need to consider what the hardware does well or poorly, how general-purpose the architecture is, and what kind of host support the hardware requires. The second consideration is the programming model that the hardware supports. Some architectures (e.g. SIMD machines) may severely limit the programming models that are practical. More general-purpose architectures will be able to support a wide variety of models, although system designers may choose to limit programmer freedom for the sake of simplicity. Other considerations include issues of program strategy, such as available expertise and the needs of planned technology demonstrations.

Although the methods used to achieve the stated goals depend on the architecture in question, we believe that the general form of the solution will be the same in all cases. We claim that any reasonable software suite for an AVIS architecture should have the following components:

- a library of image processing operators
- compilers and other tools for extending the library
- a GUI application builder
- a highly efficient runtime system and host interface

In the remainder of this section we explain what each component is, why it is needed, and what is involved in implementing it.

4.3.1 A library of image processing operators

An obvious first step toward supporting code reuse is to provide a library of reusable modules that perform commonly used operations. Our domain study suggests that (at least for low-level vision) all of the domains of interest rely on the same basic set of primitive operations. We have also found that potential AVIS users are very interested in what the systems will be able to do "out of the box", without significant programming. Khoros users whom we interviewed consistently cited

the Khoros system's large library of image and matrix operators as its most attractive feature. Thus our first recommendation is that every AVIS system be supplied with a good library of primitives.

The idea of providing a library of reusable routines is intuitively appealing and does not sound difficult. In fact it involves a number of subtle design issues, and is likely to require a great deal of work. The problems arise due to the nature of the hardware and the programming tools used to write the library routines, so we will defer discussion of these issues to the section on intermediate-level programming tools.

Library Contents

The first question to ask is what the primitive library should contain. The answer depends in part on the architecture; different designs will be suited to different types of computations, and library contents will naturally favor the things the architecture does well. All accelerators, however, should be able to handle the local, regular computations that characterize low-level vision and image processing.

Unfortunately there is no obvious comprehensive standard for image operators in any AVIS domain. The Khoros library is currently in flux owing to the release of Khoros 2.0, and is likely to emerge in a different form and with different contents from the previous version. The Image Understanding Environment (IUE) operator set has not yet been defined. Other commercial or freely available operator sets (e.g. PIKS, VISTA) lack some essential operators or facilities.

Given the lack of an obvious standard and the rapid rate of change in emerging standards and currently popular tools, we recommend the following strategy for the low-level image operator library:

1) Monitor developments in the IUE program closely. Barring unforeseen eventualities, we expect that the IUE will achieve wide acceptance in the IU and ATR communities. As soon as a specific operator set and API is chosen, consider how to achieve a reasonable level of compatibility with it.

2) In the interim, prepare to support the basic functionality described in the PIKS API (see table 2). PIKS was at one time the leading candidate for the IUE image library, and is likely to have a significant influence on the final specification. Some of the PIKS functionality (e.g. the ability for any image operator to perform subpixel shifts with resampling on any input image) is non-essential, and might be omitted on architectures for which it would restrict performance excessively.

3) In addition to PIKS functions, provide some of the missing functions identified in section 3: pyramid operators , the Canny and Wang-Binford [58] edge detectors, Moravec and other interest operators, and so on.

4) Provide a basic library of operators on non-sparse matrices and vectors. This should include (at least) vector and matrix multiplication. Routines for solving linear systems, SVD decomposition, and finding eigenvectors would be extremely useful. We recognize that some of these operations involve subtle numerical issues, and that providing a comprehensive library of matrix routines may be impractical. However, even a minimal set of matrix operators will be useful, both because of their functionality and because they will serve as examples of how to implement matrix operations efficiently.

Once we move beyond low-level image processing and mathematical operators it becomes less clear what sort of functionality the library should provide. At the intermediate level, certainly the PIKS region operators should be supported. Simple grouping operators should also be provided, including four-connected chain code extraction, the Burns and Nevatia-Babu line finders, the Hough transform, and routines that decompose curves into chains of line segments. In general, however, we do not think that there is enough consensus on what higher-level operators are useful to warrant expending a great deal of effort implementing them. Instead, system developers should focus on providing tools that will allow users to build their own intermediate and high level operators.

General-purpose architectures (e.g. MIMD machines) should be able to provide substantial support for high level vision computations via support for IUE, which defines data types and operators for a wide variety of high-level constructs. The IUE standard is very large, and producing parallelized implementations of IUE classes and methods is likely to be more work than can be accomplished under the AVIS program. However, given a good C++ compiler, MIMD architectures should be able to achieve significant parallelism even if the IUE operators consist entirely of sequential code. Many IU programs spend a lot of time iterating over sets: applying the same operator to multiple images, matching a set of image features to every model in a model base, and so on. Thus there are likely to be many opportunities for coarse-grained parallelism, which can be exploited fairly simply using a conventional language extended to support parallel iteration.

4.3.2 Tools for extending the library

It is clearly impossible to anticipate every image processing operator that domain users might need. It is also the case that many AVIS users are in the business of creating new image processing operators, and hence cannot be expected to be satisfied with a fixed image processing API. In addition to a library of basic image processing primitives, therefore, AVIS system developers will need to provide programming tools that allow users to create new library routines.

Both the nature and the role of the intermediate level programming tools depend heavily on the architecture and programming model they are intended to support. This is in contrast to the library and rapid prototyping tools, which should be relatively architecture-independent. For very general architectures such as coarse-grained MIMD machines, the same intermediate level tool set may be used both to write library routines and to build the applications that use them. Systems of this type have the advantage of allowing hierarchies of abstraction, with successive layers of software functionality being built on top of more primitive layers. This is the programming model found on conventional computers, and the one most familiar to applications programmers. For less conventional architectures, however, a hierarchical model may not be practical. Consider a system designed to support something like the PIKS API (see section 3). Here library routines would be written in a language designed specifically to take advantage of the special capabilities of the hardware. Applications would be written using the host C compiler, and would invoke library routines by sending appropriate messages to the PIKS run-time system. Hierarchical structure might exist within the library, but would not be visible to applications programs.

It is difficult to give detailed advice about what sort of intermediate-level tools AVIS system designers should provide. The decision will often hinge on subtle details of the architecture, and should be made by people who are intimately familiar with the machine in question. There is also some scope for preference, since for most architectures there are several possible solutions, each

with its own strengths and weaknesses. The basic options were described in section three. Briefly, they are:

data parallel languages These include image-specific systems such as Apply[17], as well as more general-purpose languages like Dataparallel C [18]. Most of them can be implemented efficiently on either SIMD or MIMD architectures. Some of these systems can also support MIMD-style computations via parallel iterators.

shared object systems These systems use object encapsulation to help make opportunities for parallelism visible to the run-time system and/or compiler. Parallelism is obtained by operating on multiple objects in parallel. Examples include Mentat [15] and Linda [10].

message passing systems Message passing systems such as PVM [51] and MPI [57] support a concurrent programming paradigm in which independent sequential processes communicate by sending each other messages. They are widely used for scientific computing on MIMD machines and can be very efficient in the hands of a skilled programmer. However, they are very low-level and hence are relatively difficult to use.

architecture-specific systems Truly exotic architectures may need to be programmed with tools that do not fit any standard programming paradigm. For example, configurable pipelines such as the Datacube MaxVideo system [13] or the ASPEX PIPE [23] are programmed by configuring their hardware at run time to form a sequential circuit that performs the appropriate computation. Devices based on FPGAs would be programmed similarly, perhaps with the aid of a logic assembler.

We consider the problem of designing appropriate intermediate-level tools to be the most difficult challenge facing AVIS software designers. The central question is how to provide tools that generate efficient code, are easy to use, *and* support code reuse. The ability to write efficient, reusable code is essential, since without it the library of primitives recommended above cannot be written or extended. Many current parallel programming systems have focussed exclusively on generating efficient code, and have adopted features that pose serious obstacles to code reuse. In the remainder of this section we discuss some of the issues involved and the reasons that writing reusable library code may be difficult.

Writing reusable libraries

Designing library routines for any architecture involves deciding what the fundamental abstractions should be, what functionality to provide, how to handle domain violations and exceptions and so on. These issues are not specific to parallel or exotic hardware, and we do not address them here. We are instead concerned with what makes a library routine reusable, and how implementation on a parallel machine may affect reusability.

One of the most important characteristics of reusable software is modularity or context-independence. Library routines should work correctly and efficiently regardless of the context in which they are invoked. They should neither depend on nor alter the global state of the calling program. The programmer should be able to combine and compose library routines without having to consider unintended interactions between them, and has the right to expect that the performance of each operator in isolation will be a reasonable predictor of how they will perform when used together.

For conventional machines it is not terribly difficult to write a library with these attributes. The guiding principles are well known and are now considered standard programming practice.

Functions should be 'pure' (side-effect-free). Memory and other resources should be dynamically allocated and freed when no longer needed. Function domains should be well-specified and reasonably broad, and parameters should be checked at run time. Above all, reusable library routines must avoid making assumptions about their calling context, the parameters that will be passed to them and so on, unless those assumptions are clearly spelled out in the documentation. By obeying these guidelines a good programmer can produce routines that are highly context-independent and perform well in almost any situation.

Producing reusable libraries for nonstandard architectures is far more difficult. The fundamental problem is that current methods of programming parallel architectures make it difficult to write context-independent code. Problems contributing to this situation include:

The need for tuning: Many parallel programming environments require the programmer to express decisions about data layout and granularity of parallelism in the code, either via pragmas or by means of explicit directives. The efficiency of the resulting program often depends critically on how well these decisions are made, so programmers are highly motivated to do it carefully. Unfortunately the required compiler directives have the effect of tuning the code to a particular context, making it inefficient (or even incorrect) for other arguments or in other circumstances. What is right for one situation may be wrong for another, but the library routine must work well in both.

Overhead: Many of the most common image processing operators do relatively little computation. For example, operations such as image addition or thresholding may require only a few machine instructions per pixel. On a parallel machine the cost of these operations may be dominated by the time needed to dispatch the routine. This gives the applications programmer (who probably wants to perform a whole series of image operations) a powerful motivation to abandon the library and hand-code a special-purpose routine that does the whole computation and only pays the dispatch penalty once. At this point the opportunity for code reuse has been lost and the library has failed in its purpose.

Side effects: On many parallel architectures it is impossible to write side-effect-free code. In many MIMD message-passing systems, for example, process identifiers and message tags form a global name space. Any routine that uses the messaging system risks having its messages intercepted by unrelated procedures, and may unintentionally receive messages intended for someone else. Some data-parallel languages require parallel data objects to be declared as static global variables, again creating an opportunity for name conflicts.

A particularly pernicious type of side effect arises when a library routine requires non-sharable resources. A routine that dynamically allocates temporary memory may fail if the calling program has already exhausted the heap. This can happen on sequential machines as well, but is more of a problem on parallel machines, which often lack virtual memory and tend to have small physical memories. Another example is the finite set of zero-overhead loop counters found in many DSP microprocessors. A library routine that uses a loop counter will fail when called from a program that is already using all of the loop counters.

Dedicated functional units: Some AVIS architectures may have special-purpose hardware such as convolvers, look-up tables and warping engines. These devices are generally non-sharable resources, compounding the side-effect problems described above. In addition, the operation of these devices typically depends on their internal state: the contents of their convolution kernels, LUT banks, warp function tables and so on. This is not a problem if the library routines that use

the devices establishes the appropriate state at call time. Doing so, however, may greatly increase the overhead associated with calling the routines. Using these devices efficiently may require compile-time or run-time reasoning and planning, so that (e.g.) lookup-table banks are shared whenever possible, and operations are ordered so as to minimize the number of times device state must be reloaded.

All of the problems described above were encountered in the design of the VEIL image processing system (see section 3). We found the following design principles to be useful in addressing them:

- **Manage resources internally.** Applications programmers cannot be expected to keep track of the resource requirements of library routines and call the routines only when those resources are available. Exposing resource usage in the library API creates harmful interactions between routines, destroying context-independence. It places an intolerable planning burden on the programmer, and leads to programs that are intimately tied to the architecture they were written for. In general, interactions between library routines *must* be concealed inside the library.
- **Allow for global optimization.** Making good decisions about resource allocation, scheduling and so on inherently requires a global view of the computation. The reason is that library routines for parallel machines *do* interact; the context-independent API presented to the applications programmer is an illusion. This implies that these decisions cannot be hard-coded into library routines, and in fact cannot even be made when the library is compiled. Instead they must be deferred until the application has been coded and all of its potential resource requirements are known.

In the case of VEIL, all library routines appear to be independent, although in fact they are implemented using a shared pool of memory buffers and compute engines. VEIL applications are required to declare all sequences of library operations that they might need before using any of them. The VEIL scheduler optimizes each sequence internally, so that independent operations are performed in parallel whenever possible. It also allocates resources in common across all of the sequences, arranging for lookup-table banks, memory buffers and so on to be shared appropriately. This explicit run-time scheduling step allows the system to take interactions between library routines into account, and results in executables that are tuned to run efficiently in the context of the application.

4.3.3 Support for Rapid Prototyping

Much of the work done by AVIS users in the domains covered by our study is exploratory in nature. Users need tools that allow them to experiment with different image processing and analysis strategies quickly, and to try different parameters, filter kernels and so on with minimal delay. Tools of this type are also useful as testing and debugging aids, since they make it easy for programmers to monitor and analyze intermediate stages of a computation.

Rapid prototyping tools for image processing have become extremely popular over the past few years. The most striking development in this area has been the emergence and triumph of the dataflow GUI, which is now the dominant paradigm not only in image processing, but also for scientific visualization, data acquisition, and (to a lesser extent) signal processing. Tools of this type have become a standard part of the working environment for many potential AVIS users, and are

being included as a matter of course in large-scale workstation environments such as KBVision and IUE. They should be part of the AVIS tool set as well.

All dataflow visual programming systems for image processing rely on the same basic computing paradigm, although they may differ in the details of graph semantics. Computations are represented by directed graphs whose nodes are image processing operators. Image data flows from the sources of the graph to the sinks by way of the operators, being transformed appropriately along the way. Graphs are constructed interactively using standard direct manipulation techniques, and can often be executed even before construction is complete. Section 3 of this report describes several of these environments and discusses the functionality they provide in more detail.

Although all current dataflow systems look more or less alike, we have found that they vary considerably in their usefulness as system-building tools. Many of the systems in current use evolved from scientific visualization packages such as AVS, and inherited from it a presumption that interactive exploration and visualization is the primary purpose of the tool. In some cases program graphs can be executed *only* in the context of the graphical environment. This severely limits the usefulness of the system. It allows programmers to prototype an image processing system, but provides no easy way for the prototype to evolve into a non-interactive application.

We suggest that interactive programming tools for AVIS accelerators be thought of as *application builders* rather than as tools for exploratory programming and visualization. There should be a clearly identified mechanism for incorporating a computation built using the GUI into an arbitrary application program in C, C++, or other conventional language. This capability is particularly important for researchers in real-time computer vision. They often need to write programs that perform computations on images as part of a real-time control system. In order to evaluate their image processing algorithms they must test them in the context of the control loop, so they need a rapid prototyping system that lets them move freely back and forth between the GUI and the application.

Building a dataflow rapid prototyping system

Dataflow visual programming systems can be logically divided into three main components. The most visible part of the system is the *graphical user interface* (GUI), which allows users to build a dataflow graph and interact with the computation it represents. The bulk of the computation is done by the *operator library*, a collection of programs or procedures that correspond to the nodes of the graph. Each library routine is designed to perform some primitive image processing function, or perhaps a utility function such as loading data from a file. Finally, the *run-time executive* establishes communications channels and controls the order of execution of the nodes so that the system performs the desired computation. Some systems include other components to aid with documentation, code generation and system management, but these are relatively independent of the three core subsystems.

Writing the graphical user interface for a dataflow system is not terribly difficult. For our own efforts we have found that embeddable interpreted languages such as Python [44] or tcl/tk [38] produce interfaces with acceptable performance, and allow very rapid development. In particular, the interface for Nimue, a graphical interface to VEIL [37], was written in about three weeks by a single skilled programmer.

An alternative to writing a GUI from scratch is to adopt an existing one. Khoros [24] provides explicit entry points that are intended to allow system builders to merge their own visual programming constructs into the Cantata visual language. By providing appropriate object methods, programmers can arrange for Cantata to store arbitrary information in its workspace files, along with the usual description of the graph. This would seem to offer an elegant way to integrate the functionality of AVIS accelerators and library routines into a well-known and popular development environment. This possibility will be discussed in more detail below.

Writing the code that implements the operator library should not present challenges, assuming that a good library of reusable code exists. Integrating the library into the rapid prototyping system should require no more than providing descriptions of the library routines in the form that the prototyping system requires. This information will include name, number and types of inputs and outputs, and whatever information that the executive needs in order to schedule and execute the operator.

Designing the runtime executive and execution control system is the most challenging task involved in building a dataflow visual programming system. It is also the part of the problem that depends most heavily on the architecture and intermediate-level programming model. Existing systems differ in the extent to which they attempt to determine the order of node execution by static analysis. In many cases all decision-making happens at run time. In Khoros 1.5, for example, a centralized scheduler traverses the dataflow graph at run time, using the flow of tokens along arcs of the graph to determine which nodes are eligible to be executed. Other systems compile the dataflow graph into a program for some other (possibly virtual) machine, and then hand it over to a that machine to execute. The VEIL scheduler uses this approach. In general, late decision-making allows more powerful and flexible graph semantics, while early decision-making (compilation) may result in faster programs. The Ptolemy system [7] is nearly unique in allowing users to define multiple schedulers occupying different points along the interpretation/compilation continuum.

We expect that AVIS rapid prototyping systems will make heavy use of static analysis and will compile execution schedules whenever possible, for two reasons. First, the explicit compilation step gives the system an opportunity to optimize the graph computation for the target hardware. Second, the existence of a compiled version of the dataflow graph provides an obvious path for integrating the prototype computation into a full-scale application. The challenge is to keep the compilation process fast enough to preserve a good interactive 'feel' and rapid response to user commands.

The VEIL system (see section 3) provides an example of how compiled graphs can be manipulated in an embedded application. In VEIL, dataflow graphs are ordinary C++ objects, and as such can be stored in variables, passed as parameters et cetera. Graphs can be created either procedurally (by allocating a Graph object and calling procedures that insert Operator and Link objects into the graph), or by loading them from a file. Graph files are generally created by using a GUI to construct and test the graph interactively. Once a graph has been created or loaded, the C++ program can invoke methods to

- set attributes. These methods allow the program to change operator parameters such as thresholds, window sizes and so on.
- schedule the graph. This operation traverses the graph, constructs an execution schedule, and prepares any long-lived state such as lookup tables, convolution kernels etc.

- specify synchronization. The program can attach host-side callback procedures to certain nodes, so that its own processing can be triggered by the state of the dataflow graph.
- execute the graph. The program can arrange for the graph to run once, to run continuously, or to run in alternation with other graphs.

The advantage of having applications do their image processing primarily by manipulating graphs loaded from files is that the graphs remain accessible to the GUI. If the programmer decides to change the image processing that the program is doing, he or she can halt the program, modify the graph using the GUI, and re-run the program without compiling. The program will load the modified graph and the changes will take effect immediately.

Khoros as an AVIS rapid prototyping environment

One final consideration in designing a rapid prototyping environment for AVIS systems is the social role and institutional momentum behind the Khoros system. Many ATR groups have committed to Khoros as their primary development environment, and many ATR contracts specify Khoros workspaces or modules as deliverables. The IUE committee is investigating the possibility of using Khoros as the basis of its image operator set. These facts provide a strong practical motivation for trying to use Khoros and Cantata as the rapid prototyping environment for AVIS systems.

The main obstacle that we see to building an AVIS rapid prototyping system around Khoros concerns the implementation of the runtime executive. The Khoros executive has been redesigned for version 2.0 and at present does not appear to be well localized or well documented. It does provide hooks for user-supplied execution control, but as of this writing we are unsure that there is enough flexibility to support the kind of global scheduling that AVIS systems are likely to require.

A second potential problem with using Khoros and Cantata for rapid prototyping is the lack of a good mechanism for embedding Cantata graphs into larger applications. In the current implementation of Khoros each node of a Cantata graph represents a separate UNIX executable. Provision has been made for building a shell program that executes the graph outside the context of Cantata, but the mechanism is still cumbersome and imposes tremendous overhead on the computation. AVIS application developers will need something substantially more efficient if they are to use Cantata to design image processing routines for real-time applications.

We consider integration of AVIS rapid prototyping tools into Khoros highly desirable, and probably feasible at some level. Further study is needed to determine what would be involved in providing various levels of integration, and what approach is appropriate for AVIS systems.

4.3.4 An efficient runtime system

So far we have been discussing software that will be used by programmers to build image processing applications. AVIS accelerators will also need low-level system software that provides an interface between the application and the hardware. In general this run-time software may be split between a host-side driver and an executive or operating system residing on the accelerator processor(s). The functions it provides will vary depending on the capabilities of the accelerator.

At a minimum, it will allow the host to launch operations or applications on the accelerator, send them data to be processed and retrieve the results.

Run-time software is one area where the needs of the three domains considered in this study differ considerably. The tightest constraints are those imposed by real-time computer vision. Researchers in this area use computer vision methods for tasks such as robot control and autonomous vehicle guidance. Throughput requirements for these tasks are not necessarily greater than those for general image understanding, but their latency requirements are very severe. Consider the problem of braking an autonomous vehicle to avoid a collision. In order to provide human-like competence at this task, the guidance system must be able to capture an image, analyze it, make a decision and issue a motor command in three to five hundred milliseconds. Lower level tasks such as controlling eye movements may require sampling rates of 30 to 60 frames per second, with latencies on the order of a frame time.

In order to satisfy the needs of real-time computer vision systems, run-time software for AVIS systems will need the following capabilities and attributes:

task dispatching on external events. For visual servoing applications, the accelerator will be connected directly to one or more cameras. The run-time system must allow computation to be synchronized to the video frame rate, so that computation begins as soon as the new image has been received.

low latency dispatching. Real-time vision applications need to be able to launch an accelerator computation and have it begin executing within a few hundred microseconds. For most accelerator architectures this implies that the operation of launching a computation must be kept separate from that of downloading code and large data objects (e.g. LUT contents) for the computation.

fine-grained synchronization. We assume that the host will play a significant role in most AVIS applications, even for real-time computer vision applications. The need for this is self-evident in the case of highly specialized architectures such as SIMD machines. It will also be necessary for more general-purpose architectures, if only because the application may need to talk to hardware (e.g. ethernet cards, motor controllers, file system and so on) for which the accelerator lacks device drivers. Given this assumption, it is critical that the host and accelerator be able to communicate with each other and synchronize their computations with very low overhead. Thus the run-time system should provide some mechanism for computations on either host or accelerator to issue events that can be used to synchronize access to shared resources, et cetera.

The methods used to provide and present the functionality described above depend on several factors: the intended host and host operating system, the accelerator architecture, and the model of parallelism supported by the intermediate-level programming environment. Thus it is hard to make specific recommendations about how the functionality should be provided. Fortunately the technical challenges are not severe, and in most cases there are several strategies that will work well. In the absence of other constraints we suggest supplying a *dataflow runtime system*, which we define and describe below.

Dataflow runtime systems

For the specific problem of controlling image and signal processing accelerators, there appears to be an emerging consensus about the kind of run-time system that is appropriate. The consensus is represented by the execution control facilities of the PIKS image processing API, the AT&T VCOS multimedia API, the VEIL executive, and Datacube's ImageFlow device driver. For

lack of an accepted technical term, we will refer to these as *dataflow runtime systems*. A system of this type would appear to meet all of the domain requirements identified above. In addition to image and signal processing, it would provide good support for scientific visualization, vector/matrix operations, and evaluation and training of large neural networks, assuming appropriate primitive operators were provided. It would not provide any special support for higher level computations, but neither would it interfere with them.

Dataflow runtime systems are designed to allow host applications to take advantage of attached accelerators without requiring the application programmer to write native code for them. It is implicitly assumed that the host will be responsible for most of the control flow of the application, thought there is generally some provision for decision-making on the accelerator. The primary contribution of a dataflow runtime system is to allow the application to describe complex sequences of processing steps, which the accelerator will then perform autonomously without further intervention from the application. This allows host and accelerator to run in parallel, synchronizing only when necessary.

From the applications programmer's point of view, the fundamental unit of computation is the *operator* (our term), which performs a computation on the contents of one or more memory buffers and deposits the result in other memory buffers. Operators can be grouped into *tasks*, each of which is a set of operators communicating via buffers. The runtime system allows host programs to

create buffers. This has the effect of reserving memory or other resources needed to support communications between operators, tasks and the host application.

create tasks. Tasks consist of sets of operators whose execution order may be constrained by data dependencies. Thus a task implicitly or explicitly defines a coarse-grained dataflow computation graph. For example, a task might specify that a file I/O operator should synchronize with the host to obtain images from a host-side database and deposit them in a buffer on the accelerator. The act of storing into the buffer might enable the execution of several operators that read from that buffer. Their results might be deposited in buffers read by an output operator that displays them on a monitor.

Different systems provide different mechanisms for describing tasks. In VCOS and VEIL tasks are built by creating operators and linking them into an explicit graph structure. In ImageFlow and PIKS, tasks are created by entering a special task definition mode and then issuing operator commands as if immediate execution was desired. Instead of executing the commands, the runtime system simply records them contents in the task description. The dataflow graph is defined implicitly by the use of memory buffers as input or output parameters of the commands. After exiting task definition mode, the application can dispatch the task as if it were a primitive operator, or may create other tasks.

execute a task. This has the effect of allowing the run-time system to begin executing the operators associated with the process, in the order dictated by the data dependencies and synchronization commands. In the case of PIKS, VCOS and VEIL the run-time system also takes into account the resource requirements of the tasks. ImageFlow requires the calling program to keep track of resources and avoid executing tasks with conflicting requirements in parallel.

PIKS and Imageflow allow task dispatching commands to be embedded in a task definition, so that tasks can invoke other tasks. They also support a form of conditional execution that is powerful enough to allow arbitrary conditional expressions, looping and recursion.

set operator parameters. The host application can modify an operator's behavior by sending it control information such as thresholds, gains and so on. In ImageFlow and VEIL this can be done asynchronously. In PIKS and VCOS parameter changes are entered by storing into shared buffers, and have no effect until the next invocation of an operator that reads from the buffer in question.

read or write task data. All of the systems allow the application to transfer data to or from a process by reading or writing the appropriate buffer.

synchronize with a task. All of the systems provide a mechanism that allows the application to synchronize its execution with tasks running on the accelerator. This allows the application to perform atomic updates of multiple operator parameters, implement mutual exclusion, wait for task completion, and so on. In ImageFlow synchronization is by means of *events* or semaphores. VCOS and PIKS provide the same facilities via blocking reads or writes to shared variables (*buffers* or *vregs*). VEIL supports both ImageFlow events and a PIKS-like buffering mechanism, and also allows host programs to supply callback procedures that are called automatically when the accelerator reaches a specified point in its computation.

The PIKS runtime system

The PIKS runtime system has a number of attributes that make it a good candidate for an AVIS runtime environment. PIKS is an ISO standard with a well-defined API. Based on a preliminary investigation, it appears that a number of popular systems (including both Cantata and the VEIL scheduler) could be ported to a PIKS runtime environment without much difficulty. The PIKS asynchronous execution facility also appears to be compatible with the virtual machine that underlies the Mentat distributed-object programming system [15].

Unfortunately PIKS has problems that prevent it from being adopted as is. First, it has no support for triggering on external events. It is not hard to imagine extensions that would allow a digitization operator to capture a frame from a video camera and store it in a virtual register or vreg. This would allow the arrival of the frame to trigger an arbitrary PIKS computation. At present, however, PIKS does not support this. A second problem is that PIKS operators cannot have internal state: all state information must be passed explicitly from one invocation to the next via memory buffers or vregs. This may lead to very awkward constructs for operations such as temporal FIR filtering. Finally, PIKS provides little opportunity for high level optimization and scheduling. The ISO standards document suggests performing optimizations when a task (or *chain* in PIKS terminology) is defined. However, making good decisions about how to schedule a given chain may require knowing what other chains the application requires and how it plans to use them. Without this information it may not be possible to produce good execution schedules for all applications.

At present we cannot recommend adopting PIKS as the runtime system API for AVIS accelerators. We do recommend that developers look closely at the PIKS chain and vreg mechanisms and consider providing something similar for their machines. The mechanisms are quite expressive as they stand; if they can be supported efficiently on AVIS architectures, they will be a very attractive candidate for an AVIS runtime system.

4.4 Risks and Risk Management

Software represents an area of substantial risk to the AVIS program. It will not matter how capable the hardware is if software developers cannot program it, or if the development environment is too painful to be used for exploratory programming, or if the run-time system imposes unacceptable overhead on programs. If either of the first two conditions holds, the cost of developing software for the accelerators will make them nearly useless to the communities considered in this report. If the latter condition applies, the accelerators will have difficulty competing with the rapidly advancing performance available from conventional workstations, and will be unable to meet the needs of real-time system builders.

Understanding the risks posed by software issues is the key to minimizing them. Some of the software components discussed above pose much more serious problems than others. Looking at how risks are distributed makes it possible to plan a development strategy that minimizes the chance of failure and permits design effort to be applied where it is most needed.

4.4.1 Where the risks are

By far the most serious problem posed by AVIS software requirements is that of providing good intermediate level programming tools (compilers et cetera). All of the architectures being considered for the AVIS program are complex, involving MIMD or SIMD parallelism, unusual bus structures, and/or exotic processor architectures. Automatic compilation to machines of this type is far beyond the current state of the art. Programming environments for multiprocessors are improving, but remain difficult to use and often produce very inefficient code.

The difficulty of writing programs for exotic architectures is one of the motivations for our emphasis on code reuse. If programmers can construct programs primarily by invoking reusable modules, their programs will consist mostly of flow-of-control statements and will rarely have to deal with the problems of expressing parallelism, partitioning data or allocating resources. The requirement for code reuse brings problems of its own, however. It is non-trivially harder to write a reusable convolution procedure, for example, than to write a stand-alone program for the same task. A stand-alone program can assume that it owns the entire machine, and can be carefully tuned for the image size and architecture that it is intended to run on. A reusable routine must run correctly and efficiently in any context and on any data that the programmer happens to choose.

Assuming that the intermediate-level tool problem can be solved, writing the rest of the software that AVIS systems need should be far easier. Creating a library of standard image processing operators involves no serious design issues. It will provide a good test of the intermediate-level tools, and will give developers the experience needed to create appropriate documentation and tutorials for outside programmers. At the runtime system level, the central issues relate to what constitutes a program, what role the host should play in an application, and how the host and accelerator should interact. The approach suggested above assumes a relatively active role for the host, but other solutions are possible. In all cases we believe that although this type of software is difficult to write, the technical problems are well understood and have known solutions.

Creating an environment for rapid prototyping and application building is relatively straightforward, given the excellent models provided by workstation-based environments such as Khoros. The most important design issue here is the need for a way to incorporate code built with the prototyping tools into a full-blown application. How difficult this is will depend on the archi-

tecture and on design decisions made at the run-time and intermediate levels. We claim that for heterogeneous pipelined architectures, our own VEIL system solves the problem. A similar approach should work for other architectures, although in some cases it may be necessary to provide separate tools for scheduling and resource allocation.

4.4.2 Minimizing the risks

Designing any large and complex system involves dozens of design decisions that may interact and conflict in unpredictable ways. A well-known principle of artificial intelligence states that in solving constraint satisfaction problems, the most difficult decisions should be made first. The rationale is that hard problems are less likely to have multiple solutions than easy problems, so it is better to find *some* solution to the hard problem and let it constrain the rest of the search. We believe that this principle applies to the problem of designing software for AVIS accelerators, and should be used to guide the development process. We recommend the following general strategy:

1) Choose an intermediate-level programming model. This is the most difficult problem and hence the most powerful source of constraints on subsequent design decisions. Making the choice will likely involve some prototyping of the intermediate-level tools.

2) Decide how the accelerator should look to its host and how it will interact with host programs. Making a good decision here requires bearing in mind the needs of the rapid prototyping system as well as those of more general applications programs. Part of the decision involves specifying what constitutes a 'program' for the accelerator. At one extreme, the accelerator might load a single executable file that is intended to work with (and is logically part of) a corresponding host executable. At the other end of the spectrum, the accelerator might contain a large collection of small programs, each corresponding to one or perhaps a few image processing operators. In this scenario the accelerator might act as a compute server performing remote procedure calls, perhaps from more than one host process.

3) Sketch the rest of the system: the rapid prototyping tools, intermediate-level debugger and profiler, and the rest of the run-time system. At this point standard spiral-model development can begin, with iterative refinement of the design alternating with phases of implementation and testing.

4.5 Conclusion

AVIS accelerators have the potential to transform the way research is done in the domains for which they are intended. The ability to experiment with new algorithms and to prototype applications on high-speed hardware will allow researchers to explore design space far more rapidly than in the past. This in turn should accelerate the process of applying image understanding and related technologies to problems of military and commercial interest.

Realizing the potential of AVIS systems depends as much on software as it does on hardware. If the time saved through using AVIS hardware is exceeded by the time lost due to a more difficult programming environment, researchers will stick to their familiar workstation-based tools and the accelerators will sit idle.

Providing adequate software tools for AVIS accelerators is a substantial technical challenge. The problems are tractable, however, provided that they are taken seriously and that the amount of

effort devoted to them is adequate. We hope that the information and ideas presented in this report will be of assistance to AVIS system developers, and will speed the development and distribution of advanced vision systems.

5 Bibliography

[1] ARPA. Proc. 1994 Image Understanding Workshop, Monterey, November 1994.

[2] Amerinex Artificial Intelligence, Inc., KBVision Programmers Manual, Amerinex Artificial Intelligence, Amherst, Massachusetts, 1987.

[3] M. Annaratone et al. The Warp computer: architecture, implementation and performance. IEEE Transactions on Computers v. C-36 no. 12, pp. 1523-1538, December 1987.

[4] K. Augustyn. A New Approach to Automatic Target Recognition, IEEE Trans. on Aerospace and Electronic Systems v. 28 no. 1, pp 105-114, January 1992.

[5] D. Ballard and C. Brown. Computer vision. Englewood Cliffs: Prentice-Hall, 1982.

[6] Bhanu, Bir. Automatic Target Recognition: State of the Art Survey. IEEE Transactions on Aerospace and Electronic Systems v. 22 no. 4, pp 364-379, July 1986.

[7] Buck, Joseph, Soonhoi Ha, Edward A. Lee and David G. Messerschmitt, Ptolemy: A framework for simulating and prototyping heterogeneous systems, Int. Journal of Computer Simulation (in press).

[8] Burt, Peter, and Edward H. Adelson. The Laplacian pyramid as a compact image code. IEEE Trans. on Communications v. 31 no. 4, pp. 532-540, 1983.

[9] N. Carriero and David Gelernter. How to write parallel programs: a guide for the perplexed. Computing Surveys v. 21 no. 3, pp 323-357, September 1989.

[10] N. Carriero, D. Gelernter, T.G. Mattson and A.H. Sherman. The Linda alternative to message-passing systems. Parallel Computing v. 20 no. 4, pp. 633-655, April 1994.

[11] J.L. Crowley and R.M. Stern. Fast computation for the difference of low-pass transform. IEEE Trans. Pattern Analysis and Machine Intelligence v. 6 no. 2, pp. 212-221, March 1984.

[12] Datacube, Inc., ImageFlow Reference Manual, Datacube, Inc., Danvers, Massachusetts, 1991.

[13] Datacube, Inc. MaxVideo 20 Hardware Reference Manual, Datacube, Inc., Danvers, Massachusetts, 1991.

[14] T. Eicken, D. Culler, S. Goldstein and K. Schauser. Active messages: A mechanism for integrated computation and communication. In Proc. 19th Int. Symp. on Computer Architecture. New York: ACM press, 1992.

[15] Grimshaw, Andrew S., Timothy W. Strayer, and Padmini Narayan. Dynamic, object-oriented parallel processing. IEEE Parallel and Distributed Technology v. 1 no. 2, pp. 33-47, May 1993.

[16] E. Grimson and T. Lozano-Perez. Model-based recognition and localization from sparse range or tactile data. Int. J. Robotics Research v. 3 no. 3, 1984.

[17] Leonard G. C. Hamey, John L. Webb and I-Chen Wu, Low-level vision on WARP and the APPLY programming model, in Parallel Computation and Computers for Artificial Intelligence (Janusz S. Kowalik, ed.), pp. 185-199, Kluwer Academic Publishers, Boston, 1988.

[18] Hatcher, Philip J.,Michael J. Quinn, Anthony J. Lapadula, Bradley K. Seevers, Ray J. Anderson and Robert R. Jones. Data-parallel programming on MIMD computers. IEEE Transactions on Parallel and Distributed Computing v. 3 no. 2, pp 377-381, July 1991.

[19] W.D. Hillis. The Connection Machine. Cambridge: MIT Press, 1985.

[20] Hillis, W.Daniel, and Guy L. Steele. Data parallel algorithms. Communcations of the ACM, v. 29 no. 12, pp 1170-1183, Dec. 1986.

[21] ISO. Programmer's imaging kernel system application program interface. ISO/IEC standard 12087-2:1994(E) part 2. Hackensack: ANSI, 1994.

[22] M. Kass, A. Witkin and D. Terzopoulos. Snakes: Active contour models. Int. J. Computer Vision v. 1 no. 4, pp. 321-331, Dec. 1987.

[23] E. W. Kent, M. O. Shneier, R. Lumia, PIPE - pipelined image processing engine, Journal of Parallel and Distributed Computing 2, 1985, pp 50-78.

[24] Khoral Research, Inc. KHOROS. software: available via http://www.khoros.unm.edu.

[25] Charles Kohl and Joe Mundy, The development of the Image Understanding Environ ment, in Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition, Seattle, 1994, pp. 443-447.

[26] Krueger, Charles W. Software Reuse. ACM Computing Surveys v. 24 no. 2, pp 131-184, June 1992.

[27] Michael S. Landy, Yoav Cohen and George Sperling. HIPS: A Unix-based image processing system. Computer Graphics and Image Processing v. 25, pp. 331-347, 1984.

[28] Darryl T. Lawton and Christopher C. McConnell, Image understanding environments, Proceedings of the IEEE, 76(8), August 1988, pp. 1036-1050.

[29] Edward A. Lee and David G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Transactions on Computers, v. C-36 no. 1, January 1987, pp. 24-35.

[30] Edward A. Lee and David G. Messerschmitt, Synchronous data flow, in Proceedings of the IEEE, v. 75 no. 9, September 1987, pp. 1235-1245.

[31] S. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. IEEE Trans. Pattern Analysis and Machine Intelligence v. 11 no. 7, pp. 674-693, July 1989.

[32] D. Marr. Vision. San Francisco: W.H. Freeman, 1980.

[33] O.A. McBryan. An overview of message passing environments. Parallel Computing v. 20 no. 4, pp. 417-445, April 1994.

[34] J. Mundy, T. Binford, T. Boult, A. Hanson, R. Haralick, V. Ramesh, T. Strat, C. Kohl, D. Lawton, D. Morgan, R. Beveridge, and K. Price. The Image Understanding Environment Program: IUE Class Definitions. Technical Report of the IUE committee, periodically updated, available at ftp.aai.com. October 1994.

[35] Sameer A. Nene, Shree K. Nayar, and Hiroshi Murase. SLAM: Software Library for Appearance Matching. Technical Report CUCS-019-94, Columbia University Department of Computer Science, Sept. 1994.

[36] Ramakent Nevatia and K. Ramesh Babu. Linear feature extraction and description. Computer Graphics and Image Processing v. 13 no. 3, pp. 257-269, 1980.

[37] Thomas J. Olson, Robert J. Lockwood and John R. Taylor. Programming a pipelined image processor. In Proceedings of the IEEE Workshop on Computer Architectures for Machine Perception (CAMP93), pp 93-100, New Orleans, 1993.

[38] J. K. Ousterhout, Tcl and the Tk toolkit, Addison-Wesley Publishing Co., Reading, Massachusetts, 1994.

[39] Arthur R. Pope and David G. Lowe, Vista: A software environment for computer vision research, in Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition, Seattle, 1994, pp. 768-772.

[40] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. Numerical Recipes in C. Cambridge: Cambridge University Press, 1988.

[41] John R. Rasure and Carla S. Williams, An integrated data flow visual language and software development environment, Journal of Visual Languages and Computing, v. 2, 1991, pp. 217-246.

[42] John Rasure and Steven Kubica. The Khoros application evelopment environment. In Experimental Environments for Computer Vision and Image Processing, ed. H. Christensen and J. Crowley, pp 1-32, World Scientific Press, Singapore, 1994.

[43] J. Rose and G. Steele. C*: An extended C language for data parallel programming. Tech Rep PL 87-5, Thinking Machines Corp., Cambridge, 1987.

[44] Guido van Rossum, "Interactively Testing Remote Servers Using the Python Programming Language". This paper and the Python programming language are available via anonymous ftp from ftp.cwi.nl.

[45] SAIC. Model-Driven Automatic Target Recogntion: Report of the ARPA/SAIC System Architecture Study Group. (presentation slides). October 1994.

[46] G.W. Sabot. The Paralation Model. Cambridge: The MIT Press, 1988.

[47] Schwartz, J.T., R.B.K. Dewar, E. Dubinsky, and E. Schonberg. Programming with sets: An introduction to SETL. New York: Springer-Verlag, 1986.

[48] Sipelstein, Jay M. and Guy. E. Blelloch. Collection-Oriented Languages. Proc. IEEE v. 79 no. 4, pp 504-523, April 1991.

[49] A. Skjellum, S.G. Smith, N.E. Doss, A.P. Leung and M. Morari. The design and evolution of ZipCode. Parallel Computing v. 20 no. 4, pp. 531-546, April 1994.

[50] L. Stark and K. Bowyer. Achieving generalized object recognition through reasoning about association of function to structure. IEEE Trans. Pattern Analysis and Machine Intelligence v. 13 no. 10, 1991.

[51] Sunderam, V.S. PVM: A framework for parallel distributed computing. J. Concurrency: Practice and Experience v. 2 no. 4, pp. 315-339, December 1990.

[52] M. Swain and M. Stricker, ed. Promising directions in active vision. Report of the NSF Active Vision Workshop, August 1991. Univ. of Chicago Tech Rep CS 91-27, November 1991.

[53] Tichy, Walter F., Michael Philippsen and Philip Hatcher. A Critique of the Programming Language C*. University of Karlsruhe School of Informatics Tech. Rep. 17/91, October 1991.

[54] Tucker, L.W., and A. Mainwaring. CMMD: Active messages on the CM-5. Parallel Computing v. 20 no. 4, pp. 481-496, April 1994.

[55] Matthew Turk and Alex Pentland. Eigenfaces for Recognition. J. Cognitive Neuroscience v. 3 no. 1, 1991.

[56] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, The Application Visualization System: A computational environment for scientific visualization, IEEE Computer Graphics and Applications, July 1989, pp. 30-42.

[57] D.W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. Parallel Computing v. 20 no. 4, pp. 657-673, April 1994.

[58] Wang, Sheng-Jjh, and Thomas O. Binford. Generic, model-based estimation and detection of discontinuities in image surfaces. In Proc. 1994 Image Understanding Workshop, pp. 1443-1449, Monterey, Morgan-Kaufmann Inc., November 1994.

[59] Webb, Jon A. Steps toward architecture-independent image processing. IEEE Computer v. 25 no. 2, pp. 21-31, February 1992.

[60] C. Weems, E. Riseman, A. Hanson, and A. Rosenfeld. The DARPA image understanding benchmark for parallel computers. J. Parallel and Distributed Computing v. 11 no. 1, pp. 1-24, Jan. 1991.

[61] G. Wolberg. IMPROC: An interactive image processing software package. Tech Rep CUCS-330-88, Columbia University Department of Computer Science, 1988.